Светлин Наков
Веселин Колев
и колектив

public static bool opera
(Plane one, Plane two) {
return one.TopSpeed
class Jet : Plane {
public override double
{ return 900.0D; }
public override string
{ return "I'm a Jet"; }

ВЪВЕДЕНИЕ В ПРОГРАМИРАНЕТО

със C#

telerik
deliver more than expected

БАРС
Българска асоциация
на разработчиците на софтуер

Telerik Academy

Кратко съдържание

Кратко съдържание 2
Съдържание 13
Предговор 21
Глава 1. Въведение в програмирането.....................................75
Глава 2. Примитивни типове и променливи...............................115
Глава 3. Оператори и изрази 143
Глава 4. Вход и изход от конзолата..169
Глава 5. Условни конструкции 199
Глава 6. Цикли 217
Глава 7. Масиви 241
Глава 8. Бройни системи 271
Глава 9. Методи...301
Глава 10. Рекурсия 357
Глава 11. Създаване и използване на обекти...........................391
Глава 12. Обработка на изключения.......................................421
Глава 13. Символни низове 465
Глава 14. Дефиниране на класове 509
Глава 15. Текстови файлове...625
Глава 16. Линейни структури от данни 651
Глава 17. Дървета и графи 689
Глава 18. Речници, хеш-таблици и множества 735
Глава 19. Структури от данни – съпоставка и препоръки 783
Глава 20. Принципи на обектно-ориентираното програмиране........821
Глава 21. Качествен програмен код 869
Глава 22. Ламбда изрази и LINQ заявки 925
Глава 23. Как да решаваме задачи по програмиране? 943
Глава 24. Практически задачи за изпит по програмиране – тема 1...997
Глава 25. Практически задачи за изпит по програмиране – тема 2.1051
Глава 26. Практически задачи за изпит по програмиране – тема 3.1079
Заключение...1103

Въведение в програмирането със C#

Светлин Наков, Веселин Колев и колектив

Веселин Георгиев
Веселин Колев
Дилян Димитров
Илиян Мурданлиев
Йосиф Йосифов
Йордан Павлов
Мира Бивас
Михаил Вълков
Михаил Стойнов
Николай Василев
Николай Костов
Николай Недялков
Павел Дончев
Павлина Хаджиева
Радослав Иванов
Радослав Кирилов
Радослав Тодоров
Светлин Наков
Станислав Златинов
Стефан Стаев
Теодор Божиков
Теодор Стоев
Христо Германов
Цвятко Конов

Академия на Телерик за софтуерни инженери

София, 2011

Въведение в програмирането със C#

© Академия на Телерик за софтуерни инженери,
2011 г.

Настоящата книга се разпространява свободно при следните условия:

1. Читателите **имат** право:
   - да използват книгата или части от нея за всякакви некомерсиални цели;
   - да използват сорс-кода от примерите и демонстрациите, включени към книгата или техни модификации, за всякакви нужди, включително и в комерсиални софтуерни продукти;
   - да разпространяват безплатно непроменени копия на книгата в електронен или хартиен вид;
   - да разпространяват безплатно извадки от книгата, но само при изричното споменаване на източника и авторите на съответния текст, програмен код или друг материал.
2. Читателите **нямат** право:
   - да модифицират, преправят за свои нужди или превеждат на друг език книгата без изричното съгласие на съответния автор.
   - да разпространяват срещу заплащане книгата или части от нея, като изключение прави само програмният код;

Всички запазени марки, използвани в тази книга, са собственост на техните притежатели.

Дизайн на корицата: Кристина Николова (http://krisinikolova.com/)

Официален уеб сайт:

http://www.introprogramming.info

ISBN 978-954-400-527-6

Telerik Academy

Присъедини се към Академията на Телерик!

АКАДЕМИЯТА НА ТЕЛЕРИК предоставя безплатно практическо обучение, насочено към всички млади хора, желаещи да станат умели .NET софтуерни инженери и да се присъединят към екипа на Телерик.

Академията подготвя бъдещите софтуерни специалисти за сложността, разнообразието и динамиката на днешната ИТ действителност, като за целта всички курсисти се обучават да работят с най-новите технологии на Майкрософт и черпят знания и опит от доказали се в областта на програмирането лектори и разработчици на софтуер.

В академията ще получите задълбочени знания и опит, изучавайки:

C#, .NET технологии, бази данни, SQL, Silverlight, WPF, ASP.NET, HTML5, разработка на мобилни приложения за iOS, Android и Windows Phone, основите на софтуерното инженерство

Академията на Телерик ви дава възможност да:

- **Учите напълно БЕЗПЛАТНО**
- **Изберете сред редица РАЗЛИЧНИ КУРСОВЕ**
- **Овладеете ОСНОВИТЕ на софтуерното инженерство**
- **Усвоите ПРОЦЕСА за разработка на софтуер**
- **Получите задълбочени теоретични и практически ИТ ПОЗНАНИЯ**
- **Станете умел .NET СОФТУЕРЕН ИНЖЕНЕР**
- **Започнете своята ИТ кариера в ТЕЛЕРИК – РАБОТОДАТЕЛ #1 в България за 2010 г.**

Само в рамките на две години **АКАДЕМИЯТА НА ТЕЛЕРИК** за софтуерни инженери успя да се наложи като безспорен лидер у нас в предлагането на допълнително обучение за софтуерни специалисти, спомагайки за успешния старт в кариерното развитие на стотици ентусиазирани младежи.

academy.telerik.com

academy@telerik.com *facebook.com/TelerikAcademy*

telerik
deliver more than expected

Telerik Academy SCHOOL

Стани част от Академията на Телерик за ученици!

"Основната цел на обучението е да израстнеш, а нашите умове, за разлика от телата ни, могат да растат през целия ни живот!" Мортимър Адлър

АКАДЕМИЯТА НА ТЕЛЕРИК за ученици предоставя съвременно, специализирано обучение по софтуерно инженерство за ученици от средните училища в България. Нашата основна цел е да помогнем на младите момчета и момичета да се подготвят за Националната олимпиада по информационни технологии (НОИТ) и да придобият солидни теоретични и практически умения за разработка на софтуер и софтуерни технологии.

В Училищната академия на Телерик по софтуерно инженерство:

- **Стимулираме креативността и мисленето на учениците**
- **Всеки ученик получава индивидуално внимание и съдействие**
- **Полагаме основите за писане на качествен програмен код**
- **Изучаваме съвременните софтуерни технологии**
- **Подготвяме учениците за Националната олимпиада по информационни технологии**
- **Предлагаме голямо разнообразие от практически и теоретични обучения**

Програмата на Академията включва:

C#, .NET технологии, бази данни, SQL, Silverlight, 3D графики и моделиране, разработка на игри, програмиране на микроконтролери, WPF, ASP.NET, HTML5, разработка на мобилни приложения, основи на софтуерното инженерство, работа в екип

schoolacademy.telerik.com

groups.google.com/group/it-olymp *facebook.com/TelerikSchoolAcademy*

telerik

deliver more than expected

Най-голямата ценност на компанията е екипът ѝ от млади и изпълнени с ентусиазъм специалисти

telerik
deliver more than expected

Телерик е водеща компания, предлагаща цялостни решения за разработване и автоматизирано тестване на софтуерни приложения, управление на проекти според agile методологията, бизнес репортинг, както и управление на уеб съдържание, всички от които разработени върху най-новите Microsoft платформи.

Телерик дава възможност на мотивираните, проактивни млади хора, които желаят да се развиват в ИТ индустрията, но нямат необходимите опит и знания, да се обучават безплатно в **АКАДЕМИЯТА НА ТЕЛЕРИК** и да се присъединят към екипа на компанията след успешното ѝ завършване.

Телерик е:

- **Българска иновативна технологична компания, лидер на световния пазар**
- **Златен сертифициран Microsoft партньор**
- **С екип от над 400 служители, повечето от които софтуерни инженери**
- **Работодател #1 за България през 2010 г.**
- **В челната листа на най-добрите работодатели в Централна и Източна Европа**

В Телерик ще откриете:

- **ПОСТОЯННО ОБУЧЕНИЕ И ПОМОЩ, необходими за вашето професионално развитие**
- **ДИНАМИЧНА работна среда и приятелски взаимоотношения**
- **ВЪЗМОЖНОСТ да работите заедно с едни от най-добрите софтуерни специалисти у нас**
- **ОТВОРЕНА и конструктивна комуникация**
- **УСЛОВИЯ и възможност за непрекъснато усъвършенстване**

Best Employer
Central Eastern Europe
2008/2007
conducted by Hewitt, sponsored by The Wall Street Journal Europe

Best Employer
Bulgaria
2010/2009/2008/2007
conducted by Hewitt, sponsored by The Wall Street Journal Europe

Microsoft®
GOLD CERTIFIED
Partner

Microsoft® Visual Studio®
PARTNER

www.telerik.com/careers

БАРС
Българска асоциация
на разработчиците на софтуер

www.devbg.org

Българска асоциация на разработчиците на софтуер (БАРС) е нестопанска организация, която подпомага професионалното развитие на българските софтуерни специалисти чрез образователни и други инициативи.

БАРС работи за насърчаване обмяната на опит между разработчиците и за усъвършенстване на техните знания и умения в областта на проектирането и разработката на софтуер.

Асоциацията организира специализирани конференции, семинари и курсове за обучение по разработка на софтуер и софтуерни технологии.

IT ★ BOXING

http://itboxing.devbg.org

Инициативата "IT Boxing шампионат" събира привърженици на различни софтуерни технологии и технологични доставчици в отворена дискусия на тема "коя е по-добрата технология". По време на тези събирания привърженици на двете технологии, които се противопоставят (примерно .NET и Java), защитават своята визия за по-добрата технология чрез презентации, дискусии и открит спор, който завършва с директен сблъсък с надуваеми боксови ръкавици.

Преди всяко събиране организаторите сформират две групи от експерти, които ще защитават своите технологии. Отборите презентират, демонстрират и защитават своята технология с всякакви средства. Накрая всички присъстващи гласуват и така се определя победителят.

Telerik Academy

Присъедини се към Академията на Телерик!

АКАДЕМИЯТА НА ТЕЛЕРИК предоставя безплатно практическо обучение, насочено към всички млади хора, желаещи да станат умели .NET софтуерни инженери и да се присъединят към екипа на Телерик.

Академията подготвя бъдещите софтуерни специалисти за сложността, разнообразието и динамиката на днешната ИТ действителност, като за целта всички курсисти се обучават да работят с най-новите технологии на Майкрософт и черпят знания и опит от доказали се в областта на програмирането лектори и разработчици на софтуер.

В академията ще получите задълбочени знания и опит, изучавайки:

C#, .NET технологии, бази данни, SQL, Silverlight, WPF, ASP.NET, HTML5, разработка на мобилни приложения за iOS, Android и Windows Phone, основите на софтуерното инженерство

Академията на Телерик ви дава възможност да:

- **Учите напълно БЕЗПЛАТНО**
- **Изберете сред редица РАЗЛИЧНИ КУРСОВЕ**
- **Овладеете ОСНОВИТЕ на софтуерното инженерство**
- **Усвоите ПРОЦЕСА за разработка на софтуер**
- **Получите задълбочени теоретични и практически ИТ ПОЗНАНИЯ**
- **Станете умел .NET СОФТУЕРЕН ИНЖЕНЕР**
- **Започнете своята ИТ кариера в ТЕЛЕРИК – РАБОТОДАТЕЛ #1 в България за 2010 г.**

Само в рамките на две години **АКАДЕМИЯТА НА ТЕЛЕРИК** за софтуерни инженери успя да се наложи като безспорен лидер у нас в предлагането на допълнително обучение за софтуерни специалисти, спомагайки за успешния старт в кариерното развитие на стотици ентусиазирани младежи.

academy.telerik.com

academy@telerik.com *facebook.com/TelerikAcademy*

telerik

deliver more than expected

Без съмнение това е най-добрата книга за алгоритми и програмиране, излизала на българския пазар, и ще е полезна за всички, интересуващи се от информатика.

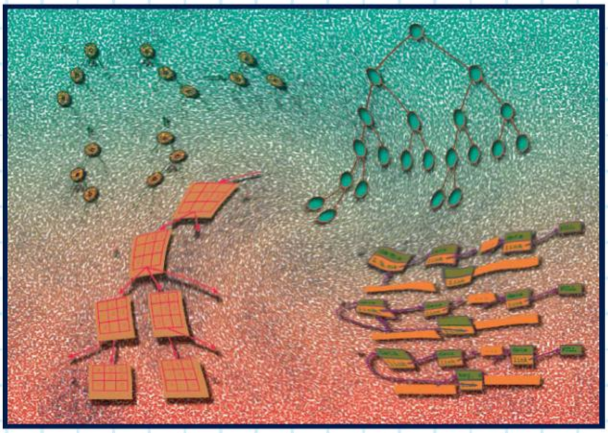
Информатически портал ИнфоМан, *http://infoman.musala.com*

Книгата, която всеки програмист трябва да притежава

Българска асоциация на разработчиците на софтуер, http://devbg.org

Програмиране=++Алгоритми;

Трето издание



Преслав Наков
Панайот Добриков

Книгата е оригинално българско творение, неотстъпващо от световното ниво в разглежданата бързо развиваща се съвременна област на компютърната информатика.
Емил Келеведжиев, Институт по математика и информатика, Българска академия на науките

За създаването на качествени програми не е достатъчно перфектното владеене на един език за програмиране. Без сериозни познания в областта на алгоритмите не е възможно да се направи ефективна програма.
доц. Красимир Манев, Факултет по Математика и Информатика, СУ "Св. Климент Охридски",
Американски университет - Благоевград

Книгата е оригинално българско творение, неотстъпващо от световното ниво в разглежданата бързо развиваща се съвременна област на компютърната информатика. Тя съчетава програмистка практика с теория, изградена върху математически методи, което допринася за по-добро разбиране и прилагане на многобройните алгоритми, съдържащи се в нея. Предназначена е за читатели, които биха оценили този синтез — ученици и техните учители, студенти и техните преподаватели, професионални програмисти и техните ръководители и разбира се, тя е за всички любители.

Книгата може да служи и като превъзходен университетски курс за въведение в алгоритмите и структурите от данни. Може да се каже, че научното и педагогическото ѝ ниво е значително. Всъщност тя е експериментирана от авторите ѝ в курса "Проектиране и анализ на компютърни алгоритми" в СУ "Св. Климент Охридски".

Забележимата разлика с повечето университетски учебници и ръководства по алгоритми и структури от данни е, че авторите използват подход "отдолу-нагоре", тръгвайки от самото програмиране, за да стигнат до теорията. Това обяснява и значителното присъствие в книгата на цялостно завършени, елегантно оформени програми с изходен текст на езика Си.

За начинаещия читател то може да служи като пътеводител в една обширна област, а за запознатия с тази област опитен програмист то има качества на справочник.

Емил Келеведжиев, Институт по математика и информатика, Българска академия на науките

За създаването на качествени програми не е достатъчно перфектното владеене на един език за програмиране. Без сериозни познания в областта на алгоритмите не е възможно да се направи ефективна програма, особено когато се работи с големи обеми от данни. За съжаление, в достъпната за българския читател литература съществува празнота в това отношение. Наличните текстове са малко и не покриват достатъчно добре темата.

Книгата, която държите в ръцете си, е следващо, по-мъдро и по-систематично, усилие след двутомното издание на единия от авторите - Преслав Наков - добре познато на всички, които в последните години са участвали в състезания по програмиране. И Преслав Наков, и Панайот Добриков са дългогодишни участници в състезания по програмиране. Отначало за ученици, а по-късно и за студенти. И двамата имат огромен опит в решаване на програмистки задачи с определено алгоритмичен характер. А многообразието от алгоритми и алгоритмични техники, много решени примери и много задачи за самостоятелна работа, са основни характеристики на настоящата книга.

Разбира се, никое издание, колкото и обемисто да е то, не може да обхване абсолютно всичко, което е направено от човечеството в областта на алгоритмите, но работата, която двамата автори са свършили, е чудесна.

доц. Красимир Манев, Факултет по Математика и Информатика, СУ "Св. Климент Охридски", Американски университет - Благоевград

Неслучайно авторите на книгата са поставили за нейно заглавие "Програмиране = ++Алгоритми;". Макар такова опростяване все повече да губи своята актуалност в съвременното програмиране, изборът на алгоритми (и структури от данни) продължава да оказва значително влияние върху ефективността на почти всяка програма.

Основният акцент в настоящата книга е поставен върху проектирането и анализа на компютърни алгоритми, независимо от конкретния език за програмиране, макар за реализацията им да е избран Си. Предложено е задълбочено и изчерпателно изложение, което се възприема лесно от читателя; едно наистина рядко съчетание.

Не може да не се подчертае, че авторите са сред най-изявените състезатели по програмиране у нас и, макар и млади, имат значителен практически опит във водещи софтуерни фирми. С тази книга на читателя се дава възможност да се запознае с важни и фини програмистки техники.

доц. Асен Рахнев, Пловдивски Университет

http://www.programirane.org/

Цена: 20 лв.

Съдържание

Кратко съдържание 2

Съдържание 13

Предговор...21

За кого е предназначена тази книга? 21

Какво обхваща тази книга?.. 23

На какво няма да ви научи тази книга? 24

Как е представена информацията?.. 24

Какво е C#?.. 25

Защо C#? 27

Примерите са върху C# 4.0 и Visual Studio 2010............................ 30

Как да четем тази книга?... 30

Защо фокусът е върху структурите от данни и алгоритмите?............ 32

Наистина ли искате ли да станете програмист? 34

За НАРС и Telerik Academy.. 36

Поглед към съдържанието на книгата... 39

За използваната терминология 47

Как възникна тази книга?... 47

Авторският колектив 50

Редакторите.. 61

Книгата е безплатна!... 62

Отзиви 62

Спонсор 71

Лиценз 72

Сайтът на книгата 74

Дискусионна група.. 74

Видеоматериали за самообучение по книгата 74

Фен клуб 74

Глава 1. Въведение в програмирането.......................................75

В тази тема... 75

Какво означава "да програмираме"?... 76

Етапи при разработката на софтуер 78

Нашата първа C# програма 82

Езикът C# и платформата .NET 85

Средата за разработка Visual Studio 2010 Express Edition................. 99

Алтернативи на Visual Studio.. 109

Декомпилиране на код 109
C# под Linux 111
Упражнения 111
Решения и упътвания 112

Глава 2. Примитивни типове и променливи 115

В тази тема... 115
Какво е променлива? 116
Типове данни 116
Променливи 128
Стойностни и референтни типове 133
Литерали 136
Упражнения 140
Решения и упътвания 141

Глава 3. Оператори и изрази 143

В тази тема... 143
Оператори 144
Преобразуване на типовете 157
Изрази 162
Упражнения 164
Решения и упътвания 165

Глава 4. Вход и изход от конзолата 169

В тази тема... 169
Какво представлява конзолата? 170
Стандартен вход-изход 173
Печатане на конзолата 174
Вход от конзолата 187
Вход и изход на конзолата – примери 192
Упражнения 194
Решения и упътвания 195

Глава 5. Условни конструкции 199

В тази тема... 199
Оператори за сравнение и булеви изрази 200
Условни конструкции if и if-else 205
Условна конструкция switch-case 210
Упражнения 213
Решения и упътвания 214

Глава 6. Цикли 217

В тази тема... 217
Какво е "цикъл"? 218
Конструкция за цикъл while 218
Конструкция за цикъл do-while 223

Конструкция за цикъл for ... 227
Конструкция за цикъл foreach ... 231
Вложени цикли ... 232
Упражнения ... 237
Решения и упътвания ... 238
Глава 7. Масиви ... 241
В тази тема... ... 241
Какво е "масив"? ... 242
Деклариране и заделяне на масиви ... 242
Достъп до елементите на масив ... 245
Четене на масив от конзолата ... 248
Отпечатване на масив на конзолата ... 250
Итерация по елементите на масив ... 251
Многомерни масиви ... 253
Масиви от масиви ... 260
Упражнения ... 263
Решения и упътвания ... 266
Глава 8. Бройни системи ... 271
В тази тема... ... 271
История в няколко реда ... 272
Бройни системи ... 273
Представяне на числата ... 283
Упражнения ... 297
Решения и упътвания ... 298
Глава 9. Методи ... 301
В тази тема... ... 301
Подпрограмите в програмирането ... 302
Какво е "метод"? ... 302
Защо да използваме методи? ... 302
Деклариране, имплементация и извикване на собствен метод ... 303
Деклариране на собствен метод ... 304
Имплементация (създаване) на собствен метод ... 308
Извикване на метод ... 310
Използване на параметри в методите ... 312
Връщане на резултат от метод ... 336
Утвърдени практики при работа с методи ... 353
Упражнения ... 354
Решения и упътвания ... 355
Глава 10. Рекурсия ... 357
В тази тема... ... 357
Какво е рекурсия? ... 358
Пример за рекурсия ... 358

Пряка и косвена рекурсия 359
Дъно на рекурсията 359
Създаване на рекурсивни методи 359
Рекурсивно изчисляване на факториел 359
Рекурсия или итерация 361
Имитация на N вложени цикъла 362
Кога да използваме рекурсия и кога итерация? 369
Използване на рекурсия – изводи 384
Упражнения 384
Решения и упътвания 386
Глава 11. Създаване и използване на обекти 391
В тази тема... 391
Класове и обекти 392
Класове в C# 394
Създаване и използване на обекти 397
Пространства от имена 411
Упражнения 416
Решения и упътвания 418
Глава 12. Обработка на изключения 421
В тази тема... 421
Какво е изключение? 422
Прихващане на изключения в C# 425
Хвърляне на изключения (конструкцията throw) 430
Йерархия на изключенията 431
Хвърляне и прихващане на изключения 433
Конструкцията try-finally 439
IDisposable и конструкцията using 444
Предимства при използване на изключения 446
Добри практики при работа с изключения 452
Упражнения 462
Решения и упътвания 463
Глава 13. Символни низове 465
В тази тема... 465
Символни низове 466
Операции върху символни низове 471
Построяване на символни низове. StringBuilder 488
Форматиране на низове 496
Упражнения 499
Решения и упътвания 504
Глава 14. Дефиниране на класове 509
В тази тема... 509
Собствени класове 510

Използване на класове и обекти 513
Съхранение на собствени класове 515
Модификатори и нива на достъп (видимост) 519
Деклариране на класове 520
Ключовата дума this 522
Полета 522
Методи 528
Достъп до нестатичните данни на класа 529
Припокриване на полета с локални променливи 532
Видимост на полета и методи 534
Конструктори 541
Свойства (Properties) 560
Статични класове (static classes) и статични членове на класа (static members) 570
Изброени типове (enumerations) 591
Вътрешни класове (nested classes) 598
Шаблонни типове и типизиране (generics) 602
Упражнения 618
Решения и упътвания 621
Глава 15. Текстови файлове 625
В тази тема... 625
Потоци 626
Четене от текстов файл 631
Писане в текстов файл 639
Обработка на грешки 641
Текстови файлове – още примери 643
Упражнения 647
Решения и упътвания 649
Глава 16. Линейни структури от данни 651
В тази тема... 651
Абстрактни структури от данни 652
Списъчни структури 653
Упражнения 683
Решения и упътвания 686
Глава 17. Дървета и графи 689
В тази тема... 689
Дървовидни структури 690
Дървета 690
Графи 723
Упражнения 730
Решения и упътвания 731
Глава 18. Речници, хеш-таблици и множества 735

В тази тема... 735
Структура от данни "речник" 736
Хеш-таблици 744
Структура от данни "множество" 770
Упражнения 778
Решения и упътвания 781
Глава 19. Структури от данни – съпоставка и препоръки 783
В тази тема... 783
Защо са толкова важни структурите данни? 784
Сложност на алгоритъм 784
Сравнение на основните структури от данни 793
Кога да използваме дадена структура? 794
Избор на структура от данни – примери 801
Външни библиотеки с .NET колекции 815
Упражнения 817
Решения и упътвания 818
Глава 20. Принципи на обектно-ориентираното програмиране 821
В тази тема... 821
Да си припомним: класове и обекти 822
Обектно-ориентирано програмиране (ООП) 822
Основни принципи на ООП 823
Наследяване (Inheritance) 823
Абстракция (Abstraction) 839
Капсулация (Encapsulation) 844
Полиморфизъм (Polymorphism) 845
Свързаност на отговорностите и функционално обвързване (cohesion и coupling) 852
Обектно-ориентирано моделиране (ООМ) 859
Нотацията UML 861
Шаблони за дизайн 863
Упражнения 867
Решения и упътвания 868
Глава 21. Качествен програмен код 869
В тази тема... 869
Защо качеството на кода е важно? 870
Какво е качествен програмен код? 870
Именуване на идентификаторите 874
Форматиране на кода 882
Висококачествени класове 890
Висококачествени методи 894
Правилно използване на променливите 899
Правилно използване на изрази 906

Използване на константи 908
Правилно използване на конструкциите за управление 910
Защитно програмиране 914
Документация на кода 917
Преработка на кода (Refactoring) 920
Ресурси 921
Упражнения 921
Решения и упътвания 922

Глава 22. Ламбда изрази и LINQ заявки 925
В тази тема... 925
Разширяващи методи (extension methods) 926
Анонимни типове (anonymous types) 928
Ламбда изрази (lambda expressions) 931
LINQ заявки (LINQ queries) 935
Упражнения 940
Решения и упътвания 941

Глава 23. Как да решаваме задачи по програмиране? 943
В тази тема... 943
Основни принципи при решаване на задачи по програмиране 944
Използвайте лист и химикал! 944
Генерирайте идеи и ги пробвайте! 945
Разбивайте задачата на подзадачи! 946
Проверете идеите си! 950
При проблем измислете нова идея! 952
Подберете структурите от данни! 954
Помислете за ефективността! 959
Имплементирайте алгоритъма си! 962
Пишете стъпка по стъпка! 963
Тествайте решението си! 976
Генерални изводи 990
Упражнения 990
Решения и упътвания 993

Глава 24. Практически задачи за изпит по програмиране – тема 1 997
В тази тема... 997
Задача 1: Извличане на текста от HTML документ 998
Задача 2: Лабиринт 1023
Задача 3: Магазин за авточасти 1036
Упражнения 1046
Решения и упътвания 1048

Глава 25. Практически задачи за изпит по програмиране – тема 2 1051

В тази тема...1051
Задача 1: Броене на думи в текст ...1052
Задача 2: Матрица с прости числа...1064
Задача 3: Аритметичен израз...1070
Упражнения...1076
Решения и упътвания ...1077
Глава 26. Практически задачи за изпит по програмиране – тема 3 ...1079
В тази тема...1079
Задача 1: Квадратна матрица ...1080
Задача 2: Броене на думи в текстов файл...1085
Задача 3: Училище...1092
Упражнения...1100
Решения и упътвания ...1101
Заключение...1103
Решихте ли всички задачи? ...1103
Имате ли трудности със задачите?...1103
На къде да продължим след книгата? ...1104
Безплатни курсове в Академията на Телерик ...1105

Предговор

Ако искате да се захванете сериозно с програмиране, попаднали сте на правилната книга. Наистина! Това е книгата, с която можете да направите първите си стъпки в програмирането. Тя ще ви даде солидни основи от знания, с които да поемете по дългия път на изучаване на съвременните езици за програмиране и технологии за разработка на софтуер. Това е книга, която учи на фундаменталните принципи на програмирането, които не са се променили съществено през последните 15 години.

Не се притеснявайте да прочетете тази книга, дори C# да не е езикът, с който искате да се занимавате. С който и друг език да продължите по-нататък, знанията, които ще ви дадем, ще ви останат трайно, защото тази книга ще ви научи да мислите като програмисти. Ще ви покажем и научим как да пишете програми, с които да решавате практически задачи по програмиране, ще ви изградим умения да измисляте и реализирате алгоритми и да ползвате различни структури от данни.

Колкото и да ви се струва невероятно, базовите принципи на писане на компютърни програми не са се променили съществено през последните 15 години. Езиците за програмиране се променят, технологиите се променят, средствата за разработка се развиват, но принципите на програмирането си остават едни и същи. Когато човек се научи да мисли алгоритмично, когато се научи инстинктивно да разделя проблемите на последователност от стъпки и да ги решава, когато се научи да подбира подходящи структури от данни и да пише качествен програмен код, тогава той става програмист. Когато придобиете тези умения, лесно можете да научите нови езици и различни технологии, като уеб програмиране, бази от данни, HTML5, XML, SQL, ASP.NET, Silverlight, Flash, Java EE и още стотици други.

Тази книга е именно за това да ви научи да мислите като програмисти, а езикът C# е само един инструмент, който може да се замени с всеки друг съвременен програмен език, например Java, C++, PHP или Python. **Това е книга за програмиране, а не книга за C#!**

За кого е предназначена тази книга?

Тази книга е най-подходяща за начинаещи. Тя е предназначена за всички, които не са се занимавали до момента сериозно с програмиране и имат желание да започнат. Тази книга стартира от нулата и ви запознава стъпка по стъпка с основните на програмирането. Тя няма да ви научи на всичко, което ви трябва, за да станете софтуерен инженер и да работите в софтуерна фирма, но ще ви даде основи, върху които да градите технологични знания и умения, а с тях вече ще можете да превърнете програмирането в своя професия.

Ако никога не сте писали компютърни програми, не се притеснявайте. Винаги има първи път. В тази книга **ще ви научим на програмиране от нулата**. Не очакваме да знаете и можете нещо предварително. Достатъчно е да имате компютърна грамотност и желание да се занимавате с програмиране. Останалото ще го прочетете от тази книга.

Ако вече можете да пишете прости програмки или сте учили програмиране в училище или в университета или сте писали програмен код с приятели, не си мислете, че знаете всичко! Прочетете тази книга и ще се убедите колко много неща сте пропуснали. Книгата е за начинаещи, но ви дава концепции, които дори някои програмисти с богат опит не владеят. По софтуерните фирми са се навъдили възмутително много самодейци, които, въпреки, че програмират на заплата от години, не владеят основите на програмирането и не знаят какво е хеш-таблица, как работи полиморфизмът и как се работи с битови операции. Не бъдете като тях! Научете първо основите на програмирането, а след това технологиите. В противен случай рискувате да останете осакатени като програмисти за много дълго време (а може би и за цял живот).

Ако пък имате опит с програмирането, за да прецените дали тази книга е за вас, я разгледайте подробно и вижте дали са ви познати всички теми, които сме разгледали. Обърнете по-голямо внимание на главите "Структури от данни – съпоставка и препоръки", "Принципи на обектно-ориентираното програмиране", "Как да решаваме задачи по програмиране?" и "Качествен програмен код". Много е вероятно дори ако имате няколко години опит, да не владеете добре работата със структури от данни, да не умеете да оценявате сложност на алгоритъм, да не владеете в дълбочина концепциите на обектно-ориентираното програмиране (включително UML и design patterns) и да не познавате добрите практики за писане на качествен програмен код. Това са много важни теми, които не се срещат във всяка книга за програмиране, така че не ги пропускайте!

Не са необходими начални познания

В тази книга не очакваме от читателите да имат предварителни знания по програмиране. Не е необходимо да сте учили информационни технологии или компютърни науки, за да четете и разбирате учебния материал. Книгата започва от нулата и постепенно ви въвлича в програмирането. Всички технически понятия, които ще срещнете, са обяснени преди това и не е нужно да ги знаете от други източници. Ако не знаете какво е компилатор, дебъгер, среда за разработка, променлива, масив, цикъл, конзола, символен низ, структура от данни, алгоритъм, сложност на алгоритъм, клас или обект, не се плашете. От тази книга ще научите всички тези понятия и много други и постепенно ще свикнете да ги ползвате непрестанно в ежедневната си работа. Просто четете книгата последователно и правете упражненията.

Ако все пак имате предварителни познания по компютърни науки и информационни технологии, при всички положения те ще са ви от полза. Ако учите университетска специалност, свързана с компютърните технологии или в училище учите информационни технологии, това само ще ви помогне, но не е задължително. Ако учите туризъм или право или друга специалност, която няма много общо с компютърните технологии, също можете да станете добър програмист, стига да имате желание.

Би било полезно да имате начална компютърна грамотност, тъй като няма да обясняваме какво е файл, какво е твърд диск, какво е мрежова карта, как се движи мишката и как се пише на клавиатурата. Очакваме да знаете как да си служите с компютъра и как да ползвате Интернет.

Препоръчва се читателите да имат някакви знания по английски език, поне начални. Всичката документация, която ще ползвате ежедневно, и почти всички сайтове за програмиране, които ще четете постоянно, са на английски език. В професията на програмиста английският е абсолютно задължителен. Колкото по-рано го научите, толкова по-добре.

> ⚠ **Не си правете илюзии, че можете да станете програмисти, без да научите поне малко английски език! Това е просто наивно очакване. Ако не знаете английски, преминете някакъв курс и след това започнете да четете технически текстове и си вадете непознатите думи и ги заучавайте. Ще се уверите, че техническият английски се учи лесно и не отнема много време.**

Какво обхваща тази книга?

Настоящата книга обхваща **основите на програмирането**. Тя ще ви научи как да дефинирате и използвате променливи, как да работите с примитивни структури от данни (като например числа), как да организирате логически конструкции, условни конструкции и цикли, как да печатате на конзолата, как да ползвате масиви, как да работите с бройни системи, как да дефинирате и използвате методи и да създавате и използвате обекти. Наред с началните познания по програмиране книгата ще ви помогне да възприемете и малко по-сложни концепции като обработка на символни низове, работа с изключения, използване на сложни структури от данни (като дървета и хеш-таблици), работа с текстови файлове и дефиниране на собствени класове и работа с LINQ заявки. Ще бъдат застъпени в дълбочина концепциите на обектно-ориентираното програмиране като утвърден подход в съвременната разработка на софтуер. Накрая ще се сблъскате с практиките за писане на висококачествени програми и с решаването на реални проблеми от програмирането. Книгата излага цялостна методология за решаване на задачи по програмиране и въобще на алгоритмични проблеми и показва как се прилага тя на практика с няколко примерни теми от изпити по програмиране. Това е нещо, което няма да срещнете в никоя друга книга за програмиране.

На какво няма да ви научи тази книга?

Тази книга няма да ви даде професията "софтуерен инженер"! Тази книга няма да ви научи да ползвате цялата .NET платформа, да работите с бази от данни, да правите динамични уеб сайтове и да боравите с прозоречен графичен потребителски интерфейс и да разработвате богати Интернет приложения (RIA). Няма да се научите да разработвате сериозни софтуерни приложения и системи като например Skype, Firefox, MS Word или социални мрежи като Facebook и търговски портали като Amazon.com. За такива проекти са нужни много, много години работа и опит и познанията от тази книга са само едно прекрасно начало.

От книгата няма да се научите на софтуерно инженерство и работа в екип и няма да можете да се подготвите за работа по реални проекти в софтуерна фирма. За да се научите на всичко това ще ви трябват още няколко книги и допълнителни обучения, но не съжалявайте за времето, което ще отделите на тази книга. Правите правилен избор като започвате от основите на програмирането вместо директно от уеб и RIA приложения и бази данни. Това ви дава шанс да станете добър програмист, който разбира програмирането и технологиите в дълбочина. След като усвоите основите на програмирането, ще ви е много по-лесно да четете и учите за бази данни и уеб приложения и ще разбирате това, което четете, много по-лесно и в много по-голяма дълбочина, отколкото, ако се захванете да учите директно SQL, ASP.NET, AJAX, WPF или Silverlight.

Някои ваши колеги започват да програмират директно от уеб приложения и бази от данни, без да знаят какво е масив, какво е списък и какво е хеш-таблица. Не им завиждайте! Те са тръгнали по трудния път, отзад напред. Ще се научат да правят нискокачествени уеб сайтове с PHP и MySQL, но ще им е безкрайно трудно да станат истински професионалисти. И вие ще научите уеб технологиите и базите данни, но преди да се захванете с тях, първо се научете да програмирате. Това е много по-важно. Да научите една или друга технология е много лесно, след като имате основата, след като можете да мислите алгоритмично и знаете как да подхождате към проблемите на програмирането.

> ⚠ **Да започнеш с програмирането от уеб приложения и бази данни е също толкова неправилно, колкото и да започнеш да учиш чужд език от някой класически роман вместо от буквар или учебник за начинаещи. Не е невъзможно, но като ти липсват основите, е много по-трудно. Възможно е след това с години да останеш без важни фундаментални знания и да ставаш за смях пред колегите си.**

Как е представена информацията?

Въпреки големия брой автори, съавтори и редактори, сме се постарали стилът на текста в книгата да бъде изключително достъпен. Съдържанието е представено в добре структуриран вид, разделено с множество заглавия и подзаглавия, което позволява лесното му възприемане, както и бързото търсене на информация в текста.

Настоящата книга е написана от програмисти за програмисти. Авторите са действащи софтуерни разработчици, колеги с реален опит както в разработването на софтуер, така и в обучението по програмиране. Благодарение на това качеството на изложението е на много добро ниво.

Всички автори ясно съзнават, че примерният сорс код е едно от най-важните неща в една книга за програмиране. Именно поради тази причина текстът е съпроводен с много, много примери, илюстрации и картинки.

Няма как, когато всяка глава е писана от различен автор, да няма разминаване между стиловете на изказ и между качеството на отделните глави. Някои автори вложиха много старание (месеци наред) и много усилия, за да станат перфектни техните глави. Други не вложиха достатъчно усилия и затова някои глави не са така хубави и изчерпателни като другите. Не на последно място опитът на авторите е различен: някои програмират професионално от 2-3 години, докато други – от 15 години насам. Няма как това да не се отрази на качеството, но ви уверяваме, че всяка глава е минала редакция и отговаря поне минимално на високите изисквания на водещите автори на книгата – Светлин Наков и Веселин Колев.

Какво е C#?

Вече обяснихме, че тази книга не е за езика C#, а за програмирането като концепция и неговите основни принципи. Ние използваме езика C# и платформата Microsoft .NET Framework само като средство за писане на програмен код и не наблягаме върху спецификите на езика. Настоящата книга може да бъде намерена и във варианти за други езици като Java и C++, но разликите не са съществени.

Все пак, нека разкажем с няколко думи какво е C# (чете се "**си шарп**").

C# е съвременен език за програмиране и разработка на софтуерни приложения.

Ако думичките "C#" и ".NET Framework" ви звучат непознато, в следващата глава ще научите подробно за тях и за връзката между тях. Сега нека все пак обясним съвсем накратко какво е C#, какво е .NET, .NET Framework, CLR и останалите свързани с езика C# технологии.

Езикът C#

C# е съвременен обектно-ориентиран език за програмиране с общо предназначение, създаден и развиван от Microsoft редом с .NET платформата. На езика C# и върху .NET платформата се разработва изключително разнообразен софтуер: офис приложения, уеб приложения и уеб сайтове, настолни приложения, богати на функционалност мултимедийни Интернет приложения (RIA), приложения за мобилни телефони, игри и много други.

C# е език от високо ниво, който прилича на Java и C++ и донякъде на езици като Delphi, VB.NET и C. Всички C# програми са обектно-ориентирани. Те представляват съвкупност от дефиниции на класове, които съдържат в себе си методи, а в методите е разположена програмната логика – инструкциите, които компютърът изпълнява. Повече детайли за това какво е клас, какво е метод и какво представляват C# програмите ще научите в следващата глава.

В днешно време C# е един от най-популярните езици за програмиране. На него пишат милиони разработчици по цял свят. Тъй като C# е разработен от Майкрософт като част от тяхната съвременна платформа за разработка и изпълнение на приложения .NET Framework, езикът е силно разпространен сред Microsoft-ориентираните фирми, организации и индивидуални разработчици. За добро или за лошо към момента на написване на настоящата книга езикът C# и платформата .NET Framework се поддържат и контролират изцяло от Microsoft и не са отворени за външни участници. По тази причина всички останали големи световни софтуерни корпорации като IBM, Oracle и SAP базират своите решения на Java платформата и използват Java като основен език за разработка на своите продукти.

За разлика от C# и .NET Framework езикът и платформата Java са проекти с отворен код, в които участва цялата общност от софтуерни фирми, организации и индивидуални разработчици. Стандартите, спецификациите и всички новости в Java света се развиват чрез работни групи, съставени от цялата Java общност, а не от една единствена фирма (както е със C# и .NET Framework).

Езикът C# се разпространява заедно със специална среда, върху която се изпълнява, наречена **Common Language Runtime (CLR)**. Тази среда е част от **платформата .NET Framework**, която включва CLR, пакет от стандартни библиотеки, предоставящи базова функционалност, компилатори, дебъгери и други средства за разработка. Благодарение на нея CLR програмите са преносими и след като веднъж бъдат написани, могат да работят почти без промени върху различни хардуерни платформи и операционни системи. Най-често C# програмите се изпълняват върху MS Windows, но .NET Framework и CLR се поддържа и за мобилни телефони и други преносими устройства базирани на Windows Mobile. Под Linux, FreeBSD, MacOS X и други операционни системи C# програмите могат да се изпълняват върху свободната .NET Framework имплементация Mono, която обаче не се поддържа официално от Microsoft.

Платформата Microsoft .NET Framework

Езикът C# не се разпространява самостоятелно, а е част от платформата Microsoft .NET Framework (чете се "**майкрософт дот нет фреймуърк**"). .NET Framework най-общо представлява среда за разработка и изпълнение на програми, написани на езика C# или друг език, съвместим с

.NET (като VB.NET, Managed C++, J# или F#). Тя се състои от .NET езици за програмиране (C#, VB.NET и други), **среда за изпълнение на управляван код (CLR)**, която изпълнява контролирано C# програмите, и от съвкупност от стандартни библиотеки и инструменти за разработка, като например компилаторът `csc`, който превръща C# програмите в разбираем за CLR междинен код (наречен MSIL) и библиотеката ADO.NET, която осигурява достъп до бази от данни (например MS SQL Server или MySQL). .NET Framework е част от всяка съвременна Windows дистрибуция и може да се срещне в различни свои версии. Последна версия може да се изтегли и инсталира от сайта на Microsoft. Към момента на публикуването на настоящата книга най-новата версия на .NET Framework е 4.0, а стандартно във Windows Vista е включен .NET Framework 2.0, а в Windows 7 – версия 3.5.

Защо C#?

Има много причини да изберем езика C# за нашата книга. Той е съвременен език за програмиране, широкоразпространен, използван от милиони програмисти по целия свят. Същевременно C# е изключително прост и лесен за научаване език (за разлика от C и C++). Нормално е да започнем от език, който е подходящ за начинаещи и се ползва много в практиката. Именно такъв език избрахме – лесен и много популярен, език, който се ползва широко в индустрията от много големи и сериозни фирми.

C# или Java?

Въпреки, че по този въпрос може много да се спори, се счита, че единственият сериозен съперник на C# е Java. Няма да правим сравнение между Java и C#, тъй като C# безспорно е по-добрият, по-мощният и по-добре измисленият от инженерна гледна точка език, но трябва да обърнем внимание, че за целите на настоящата книга всеки съвременен език за програмиране ще ни свърши работа. Ние избрахме C#, защото е по-лесен за изучаване и се разпространява с изключително удобни безплатни среди за разработка (например Visual C# Express Edition). Който има предпочитания към Java може да ползва Java варианта на настоящата книга, достъпен от нейния сайт: www.introprogramming.info.

Защо не PHP?

По отношение на популярност освен C# и Java много широко използван е езикът PHP. Той е подходящ за разработка на малки уеб сайтове и уеб приложения, но създава сериозни трудности при реализацията на големи и сложни софтуерни системи. В софтуерната индустрия PHP се ползва предимно за малки проекти, тъй като предразполага към писане на лош, неорганизиран и труден за поддръжка код, поради което е неудобен за по-сериозни проекти. По този въпрос може много да се спори, но се счита, че поради остарелите си концепции и подходи, върху които е построен, и поради редица еволюционни причини PHP е език, който предразполага към некачествено програмиране, писане на лош код и изграждане на труден за поддръжка софтуер. PHP е по идея процедурен език и въпреки че поддържа парадигмите на съвременното обектно-ориентирано програмиране повечето PHP програмисти пишат процедурно. PHP е известен като езикът на "мазачите" в професията на софтуерните разработчици, защото повечето PHP програмисти пишат ужасно некачествен код. Поради склонността да се пише некачествен, лошо структуриран и лошо организиран програмен код цялата концепция на езика и платформата PHP се счита са сгрешена и сериозните фирми (като например Microsoft, SAP и Oracle и техните партньори) я отбягват. По тази причина, ако искате да станете сериозен софтуерен инженер, започнете от C# или Java и избягвайте PHP (доколкото е възможно).

PHP си има своето приложение в света на програмирането (например да си направим блог с WordPress, малък сайт с Joomla или Drupal или дискусионен форум с PhpBB), но цялата PHP платформа не е така зряла и добре организирана като .NET и Java. Когато става дума за не-уеб базирани приложения или големи индустриални проекти, PHP изобщо не е сред възможностите за избор. За да се ползва коректно PHP и да се разработват професионални проекти с високо качество е необходим много, много опит. Обикновено PHP разработчиците учат от самоучители, статии и книги с ниско качество и заучават вредни практики и навици, които след това е много трудно да се изчистят. Затова не учете PHP като ваш пръв език за разработка. Започнете със C# или Java.

На базата на огромния опит на авторския колектив можем да ви препоръчаме да започнете да учите програмиране с езика C# като пропуснете езици като C, C++ и PHP до момента, в който не ви се наложи да ги ползвате.

Защо не C или C++?

Въпреки, че отново много може да се спори, езиците C и C++ се считат за доста примитивни, остарели и отмиращи. Те все пак имат своето приложение и са подходящи за програмиране на ниско ниво (например за специализирани хардуерни устройства), но не ви съветваме да се занимавате с тях.

На **чисто C** може да програмирате, ако трябва да пишете операционна система, драйвер за хардуерно устройство или да програмирате промишлен контролер (embedded device), поради липса на алтернатива и поради нуждата да се управлява много внимателно хардуера. Този език е морално остарял и в никакъв случай не ви съветваме да започвате да учите програмиране с него. Производителността на програмиста при разработка на чисто C е в пъти по-ниска отколкото при съвременните езици за програмиране с общо предназначение като C# и Java. Вариант на езика C се използва при Apple / iPhone разработчиците, но не защото е хубав език, а защото няма свястна алтернатива. Повечето Apple-ориентирани разработчици не харесват Objective-C, но нямат избор да пишат на нещо друго.

C++ е добър, когато трябва да програмирате определени приложения, които изискват много близка работа с хардуера или имат специални изисквания за бързодействие (например разработка на 3D игри). За всичко станали задачи (например разработка на уеб приложения или бизнес софтуер) C++ е изключително неподходящ. Не ви съветваме да се захващате с него, ако сега стартирате с програмирането. Причината все още да се учи C++ в някои училища и университети е наследствена, тъй като тези институции са доста консервативни. Например международната олимпиада по информатика за ученици (IOI) продължава да промоцира C++ като единствения език, позволен за използване по състезанията по програмиране, въпреки, че в индустрията C++ почти не се използва. Ако не вярвате, разгледайте някой сайт с обяви и пребройте колко процента от обявите за работа изискват C++.

Езикът C++ изгуби своята популярност най-вече поради невъзможността на него да се разработва бързо качествен софтуер. За да пишете кадърно на C++, трябва да сте много печен и опитен програмист, докато за C# и Java не е чак толкова задължително. Ученето на C++ отнема в пъти повече време и много малко програмисти го владеят наистина добре. Производителността на C++ програмистите е в пъти по-ниска от C# и затова C++ все повече губи позиции. Поради всички тези причини, този език постепенно си отива и затова не ви съветваме да го учите.

Предимствата на C#

C# е обектно-ориентиран език за програмиране. Такива са всички съвременни езици, на които се разработват сериозни софтуерни системи (например Java и C++). За предимствата на обектно-ориентираното програмиране (ООП) ще стане дума подробно на много места в книгата, но за момента може да си представяте обектно-ориентираните езици като езици, които позволяват да работите с обекти от реалния свят (примерно студент, училище, учебник, книга и други). Обектите имат характеристики (примерно име, цвят и т.н.) и могат да извършват действия (примерно да се движат, да говорят и т.н).

Започвайки с програмирането от езика C# и платформата .NET Framework вие поемате по един много перспективен път. Ако отворите някой сайт с обяви за работа за програмисти, ще се убедите, че търсенето на C# и .NET специалисти е огромно и е близко до обема на търсенето на Java програмисти. Същевременно търсенето на специалисти по PHP, C++ и всички останали технологии е много по-малко отколкото търсенето на C# и Java инженери.

За добрия програмист езикът, на който пише, няма съществено значение, защото той умее да програмира. Каквито и езици и технологии да му трябват, той бързо ги овладява. Нашата цел е не да ви научим на C#, а да ви научим на програмиране! След като овладеете основите на програмирането и се научите да мислите алгоритмично, можете да научите и други езици и ще се убедите колко много приличат те на C#. Програмирането се гради на принципи, които много бавно се променят с годините и тази книга ви учи точно на тези принципи.

Примерите са върху C# 4.0 и Visual Studio 2010

Всички примери в книгата се отнасят за версия 4.0 на езика C# и платформата .NET Framework, която към момента на публикуване на книгата е последната. Всички примери за използване на средата за разработка Visual Studio се отнасят за версия 2010 на продукта, която също е последна към момента на публикуване на книгата.

Средата за разработка Microsoft Visual Studio 2010 има безплатна версия, подходяща за начинаещи C# програмисти, наречена Microsoft Visual C# 2010 Express Edition, но разликата между Express Edition и пълната версия на Visual Studio (която е комерсиален софтуерен продукт), е във функции, които няма да са ни необходими в рамките на тази книга.

Въпреки, че използваме C# 4.0 и Visual Studio 2010, сме се постарали да не използваме екзотичните новости на C# 3.0 и 4.0 за да не предизвикваме объркване в читателя. Реално почти всички примери, които ще намерите в тази книга, работят безпроблемно под .NET Framework 2.0 и C# 2.0 и могат да се компилират с Visual Studio 2005.

Коя версия на C# и Visual Studio ще ползвате докато се учите да програмирате не е от съществено значение. Важното е да научите принципите на програмирането и алгоритмичното мислене! Езикът C#, платформата .NET Framework и средата Visual Studio са само едни инструменти и можете да ги замените с други по всяко време.

Как да четем тази книга?

Четенето на тази книга трябва да бъде съпроводено с много, **много практика**. Няма да се научите да програмирате, ако не практикувате! Все едно да се научите да плувате от книга, без да пробвате. Няма начин! Колкото повече работите върху задачите след всяка глава, толкова повече ще научите от книгата.

Всичко, което прочетете тук, трябва да изпробвате сами на компютъра. Иначе няма да научите нищо. Примерно, когато прочетете за Visual Studio и как да си направите първата проста програмка, трябва непременно да си изтеглите и инсталирате Microsoft Visual Studio (или Visual C# Express) и да пробвате да си направите някаква програмка. Иначе няма да се научите! На теория винаги е по-лесно, но програмирането е практика. Запомнете това и решавайте задачите за упражнения от книгата. Те са внимателно подбрани – хем не са много трудни, за да не ви откажат, хем не са много лесни, за да ви мотивират да приемете решаването им като предизвикателство. Ако имате трудности, потърсете помощ в дискусионната група на курса "C# Fundamentals", който се води по тази книга: http://groups.google.com/group/telerikacademy/.

Четенето на тази книга без практика е безсмислено! Трябва да отделите за писане на програми няколко пъти повече време, отколкото отделяте да четете текста.

Всеки е учил математика в училище и знае, че за да се научи да решава задачи по математика, му трябва много практика. Колкото и да гледа и да слуша учителя, без да седне да решава задачи никой не може да се научи. Така е и с програмирането. Трябва ви много практика. Трябва да пишете много, да решавате задачи, да експериментирате, да се мъчите и да се борите с проблемите, да грешите и да се поправяте, да пробвате и да не става, да пробвате пак по нов начин и да изживеете моментите, в които се получава. Трябва ви много, много практика. Само така ще напреднете.

Не пропускайте упражненията!

На края на всяка глава има сериозен списък със задачи за упражнения. Не ги пропускайте! Без упражненията нищо няма да научите. След като прочетете дадена глава, трябва да седнете на компютъра и да пробвате примерите, които сте видели в книгата. След това трябва да се хванете и да решите всички задачи. Ако не можете да решите всички задачи, трябва поне да се помъчите да го направите. Ако нямате време, трябва да решите поне първите няколко задачи от всяка глава. Не преминавайте напред, без да решавате задачите след всяка глава! Просто няма смисъл. Задачите са малки реални ситуации, в които прилагате прочетеното. В практиката, един ден, когато станете програмисти, ще решавате всеки ден подобни задачи, но по-големи и по-сложни.

Непременно решавайте задачите за упражнения след всяка глава от книгата! Иначе рискувате нищо да не научите и просто да си загубите времето.

Колко време ще ни трябва за тази книга?

Усвояването на основите на програмирането е много сериозна задача и отнема много време. Дори и силно да ви се отдава, няма начин да се научите да програмирате на добро ниво за седмица или две. За научаването на всяко човешко умение е необходимо да прочетете или да видите или да ви покажат как се прави и след това да пробвате сами. При програмирането е същото – трябва или да прочетете или да гледате или да слушате как се прави, след това да пробвате сами и да успеете или да не успеете и да пробвате пак и накрая да усетите, че сте го научили. Ученето става стъпка по стъпка, последователно, на малки порции, с много усърдие и постоянство.

Ако искате да прочетете, разберете, научите и усвоите цялостно и в дълбочина целия учебния материал от тази книга, ще трябва да инвестирате поне 2 месеца целодневно или поне 4-5 месеца, ако четете и се упражнявате по малко всеки ден. Това е минималното време, за което можете да усвоите в дълбочина основните на програмирането. Нуждата от такъв обем занимания е потвърдена от безплатните обучения в Академията на Телерик (http://academy.telerik.com), които се водят точно по тази книга. Стотиците курсисти, преминали обучение по лекциите към книгата, обикновено научават за 3 месеца целодневно всички теми от тази книга, решават всички задачи за упражнения и полагат успешно изпити по учебния материал. Статистиката сочи, че всеки, който отдели за тази книга и съответния курс в Академията на Телерик по-малко време от равностойността на 3 месеца целодневно, без да има солидни предварителни знания по програмиране, се проваля на изпитите.

Основният учебен материал в книгата е изложен в около 1100 страници, за които ще ви трябват около месец (по цял ден) само, за да го прочетете внимателно и да изпробвате примерните програми. Разбира се, трябва да отделите достатъчно внимание и на упражненията, защото без тях почти нищо няма да научите.

Упражненията съдържат около 350 задачи с различна трудност. За някои от тях ще ви трябват по няколко минути, докато за други ще ви трябват по няколко часа (ако въобще успеете да ги решите без чужда помощ). Това означава, че ще ви трябва месец-два по цял ден да се упражнявате или да го правите по малко в продължение на няколко месеца.

Ако не разполагате с толкова време, замислете се дали наистина искате да се занимавате с програмиране. Това е много сериозно начинание, в което трябва да вложите наистина много усилия. Ако наистина искате да се научите да програмирате на добро ниво, планувайте си достатъчно време и следвайте книгата.

Защо фокусът е върху структурите от данни и алгоритмите?

Настоящата книга наред с основните познания по програмиране ви учи и на правилно алгоритмично мислене и работа с основните структури от данни в програмирането. Структурите от данни и алгоритмите са най-важните фундаментални знания на един програмист! Ако ги овладеете добре, след това няма да имате никакви проблеми да овладеете която и да е софтуерна технология, библиотека, framework или API. Именно на това разчитат и най-сериозните софтуерни фирми в света, когато наемат служители. Потвърждение са интервютата в големите фирми като Google и Microsoft, които изключително много държат на правилното алгоритмично мислене и познаването на всички базови структури от данни и алгоритми.

Интервютата за работа в Google

На интервютата за работа като софтуерен инженер в Google в Цюрих 100% от въпросите са съсредоточени върху структури от данни, алгоритми и алгоритмично мислене. На такова интервю могат да ви накарат да реализирате на бяла дъска свързан списък (вж. главата "Линейни структури от данни") или да измислите алгоритъм за запълване на растерен многоъгълник (зададен като GIF изображение) с даден цвят (вж. метод на вълната в главата "Дървета и графи". Изглежда Google ги интересува да наемат хора, които имат алгоритмично мислене и владеят основните структури от данни и базовите компютърни алгоритми. Всички технологии, които избраните кандидати ще използват след това в работата си, могат бързо да бъдат усвоени. Разбира се, не си мислете, че тази книга ще ви даде всички знания и умения, за да преминете успешно интервю за работа в Google. Знанията от книгата са абсолютно необходими, но не са достатъчни. Те са само първите стъпки.

Интервютата за работа в Microsoft

На интервютата за работа като софтуерен инженер в Microsoft в Дъблин голяма част от въпросите са съсредоточени върху структури от данни, алгоритми и алгоритмично мислене. Например могат да ви накарат да обърнете на обратно всички думи в даден символен низ (вж. главата "Символни низове") или да реализирате топологично сортиране в неориентиран граф (вж. главата "Дървета и графи"). За разлика от Google в Microsoft питат и за много инженерни въпроси, свързани със софтуерни архитектури, паралелна обработка (multithreading), писане на сигурен код, работа с много големи обеми от данни и тестване на софтуера. Настоящата книга далеч не е достатъчна, за да кандидатствате в Microsoft, но със сигурност знанията от нея ще ви бъдат полезни за една голяма част от въпросите.

За технологията LINQ

В книгата е включена една тема за популярната технология LINQ (Language Integrated Query), която позволява изпълнение на различни заявки (като търсене, сортиране, сумиране и други групови операции) върху масиви, списъци и други обекти. Тя нарочно е разположена към края, след темите за структури от данни и сложност на алгоритми. Причината за това е, че добрият програмист трябва да знае какво се случва, когато сортира списък или търси по даден критерий в масив и колко операции отнемат тези действия. Ако се използва LINQ, не е очевидно как работи дадена заявка и колко време отнема. LINQ е много мощна и широко-използвана технология, но тя трябва да бъде овладяна на по-късен етап, след като познавате добре основите на програмирането и основните алгоритми и структури от данни. В противен случай рискувате да се научите да пишете неефективен код без да си давате сметка как работи той и колко операции извършва.

Наистина ли искате ли да станете програмист?

Ако искате да станете програмист, трябва да знаете, че **истинските програмисти** са сериозни, упорити, мислещи и търсещи хора, които се справят с всякакви задачи, и за тях е важно да могат бързо да овладяват всички необходими за работата им платформи, технологии, библиотеки, програмни средства, езици за програмиране и инструменти за разработка и да усещат програмирането като част от себе си.

Добрите програмисти отделят изключително много време да развиват инженерното си мисленето, да учат ежедневно нови технологии, нови езици за програмиране, нови начини на работа, нови платформи и нови средства за разработка. Те умеят да мислят логически да разсъждават върху проблемите и да измислят алгоритми за решаването им, да си представят решенията като последователност от стъпки, да моделират заобикалящия ги свят със средствата на технологиите, да реализират идеите си като програми или програмни компоненти, да тестват алгоритмите и програмите си, да виждат проблемите, да предвиждат изключителните ситуации, които могат да възникнат и да ги обработват правилно, да се вслушват в съветите на по-опитните от тях, да съобразяват потребителския интерфейс на програмите си с потребителя и да съобразяват алгоритмите си с възможностите на машините, върху които те се изпълняват, и със средата, в която работят и с която си взаимодействат.

Добрите програмисти непрекъснато четат книги, статии или блогове за програмиране и се интересуват от новите технологии, постоянно обогатяват познанията си и подобряват начина на работата си и качеството на написания от тях софтуер. Някои от тях се вманиачават до такава степен, че забравят да ядат или спят, когато имат да решат сериозен проблем или просто са вдъхновени от някоя нова технология или гледат някоя интересна лекция или презентация. Ако вие имате склонност да се мотивирате до такава степен в едно нещо (например да играете денонощно компютърни игри), можете много бързо да се научите да програмирате, стига просто да се настроите, че най-интересното нещо на света за вас в този период от живота ви е програмирането.

Добрите програмисти имат един или няколко компютъра и Интернет и живеят в постоянна връзка с технологиите. Те посещават редовно сайтове и блогове, свързани с новите технологии, комуникират ежедневно със свои колеги, посещават технологични лекции, семинари и други събития, следят технологичните промени и тенденции, пробват новите технологии, дори и да нямат нужда от тях в момента, експериментират и проучват новите възможности и новите начини да направят даден софтуер или елемент от работата си, разглеждат нови библиотеки, учат нови езици, пробват нови технологични рамки (frameworks) и си играят с новите средства за разработка. Така те развиват своите умения и поддържат доброто си ниво на информираност, компетентност и професионализъм.

Истинските програмисти знаят, че никога не могат да овладеят професията си до краен предел, тъй като тя се променя постоянно. Те живеят с твърдото убеждение, че цял живот трябва да учат и това им харесва и им носи удовлетворение. Истинските програмисти са любопитни и търсещи хора, които искат да знаят всичко как работи – от обикновения аналогов часовник, до GPS системата, Интернет технологиите, езиците за програмиране, операционните системи, компилаторите, компютърната графика, игрите, хардуера, изкуствения интелект и всичко останало, свързано с компютрите и технологиите. Колкото повече научават, толкова повече са жадни за още знания и умения. Животът им е свързан с технологиите и те се променят заедно с тях, наслаждавайки се на развитието на компютърните науки, технологиите и софтуерната индустрия.

Всичко, за което ви разказваме за истинските програмисти, го знаем от личен опит и сме убедени, че програмист е професия, която иска да й се посветиш и да й се отдадеш напълно, за да бъдеш наистина добър специалист – опитен, компетентен, информиран, мислещ, разсъждаващ, знаещ, можещ, умеещ да се справя в нестандартни ситуации. Всеки, който се захване с програмиране "помежду другото" е обречен да бъде посредствен програмист. Програмирането изисква пълно себеотдаване в продължение на години. Ако сте готови за всичко това, продължавайте да четете напред и си дайте сметка, че тези няколко месеца, които ще отделите на тази книга за програмиране са само едно малко начало, а след това години наред ще учите докато превърнете програмирането в своя професия, а след това ежедневно ще учите по нещо и ще се състезавате с технологиите, за да поддържате нивото си, докато някой ден програмирането ви даде достатъчно развитие на мисленето и уменията ви, за да се захванете с друга професия, защото са малко програмистите, които програмират до пенсия, но са наистина много успешните хора, стартирали кариерата си с програмиране.

Ако все още не сте се отказали да станете добър програмист и вече сте си изградили дълбоко в себе си разбиране, че следващите месеци и години от живота си ще бъдат ежедневно свързани с постоянен усърден труд по овладяване на тайните на програмирането, разработката на софтуер, компютърните науки и софтуерните технологии, може да използвате една стара техника за вътрешна мотивация и уверено постигане на цели, която може да бъде намерена в много книги и древни учения под една или друга форма: представяйте си постоянно, че сте програмисти, че сте успели да станете такива, че се занимавате ежедневно с програмиране и то е вашата професия и че можете да напишете целия софтуер на света (стига да имате достатъчно време), че умеете да решите всяка задача, която опитните програмисти могат да решат, и си мислете постоянно и непрекъснато за вашата цел. Казвайте на себе си, дори понякога на глас: "аз искам да стана добър програмист и трябва много да работя за това, трябва много да чета и много да уча, трябва да решавам много задачи, всеки ден, постоянно и усърдно". Сложете книгите за програмиране навсякъде около вас, дори залепете надпис "аз ще стана добър програмист" над леглото си, така че да го виждате всяка вечер, когато лягате и всяка сутрин, когато ставате. Програмирайте всеки ден, решавайте задачи, забавлявайте се, учете нови технологии, експериментирайте, пробвайте да напишете игра, да направите уеб сайт, да напишете компилатор, база данни и още стотици програми, за които ви хрумнат оригинални идеи. За да станете добри програмисти програмирайте всеки ден и всеки ден мислете за програмирането и си представяйте бъдещия момент, в който вие сте отличен програмист. Можете, ако дълбоко вярвате, че можете. Всеки може, стига да вярва, че може и да следва целите си постоянно, без да се отказва. Никой не може да ви мотивира по-добре от вас самите. Всичко зависи от вас и тази книга е първата ви стъпка.

За НАРС и Telerik Academy

Тъй като водещият автор на книгата Светлин Наков (www.nakov.com) е създател на добре известната в софтуерната индустрия Национална академия по разработка на софтуер (НАРС) и на инициативата Telerik Academy на софтуерната корпорация Телерик, си позволяваме да обясним връзката между Светлин Наков, двете академии и настоящата книга.

Национална академия по разработка на софтуер (НАРС)

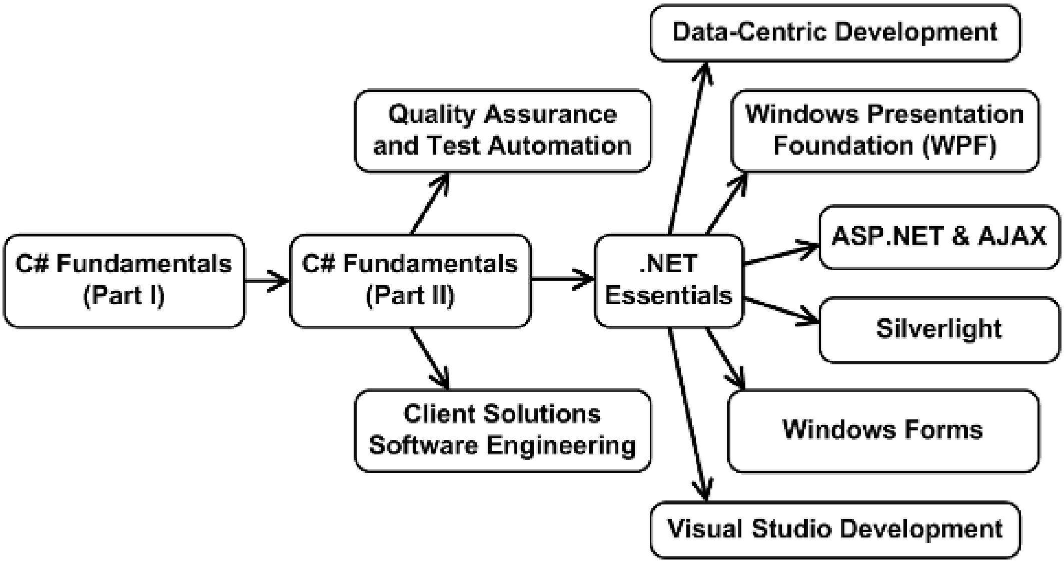
Първоначално книгата "Въведение в програмирането с Java" (предшественик на настоящата книга) възниква като проект на НАРС под ръководството на Светлин Наков. По това време НАРС е във възход и осигурява безплатно обучение и работа в сферата на софтуерното инженерство на над 600 души, млади хора, предимно студенти. Академията провежда безплатни курсове за подготовка на квалифицирани специалисти за големи софтуерни фирми като SAP, Telerik, Johnson Controls (JCI), VMWare, Euro Games Technology (EGT), Musala Soft, Stemo, Rila Solutions, Sciant (VMware Bulgaria), Micro Focus, InterConsult Bulgaria (ICB), Acsior, Fadata, Seeburger Informatik и др. НАРС организира безплатни едномесечни курсове по "Въведение в програмирането", които обхващат в голяма степен учебния материал от настоящата книга, след което най-добрите от обучените курсисти продължават да учат безплатно още 5 месеца по стипендии от фирми, които им осигуряват работа по специалността след успешно завършване. Този модел на обучение работеше преди финансовата криза през 2009 г., но когато тя обхвана цялата софтуерна индустрия постепенно НАРС загуби позиции и нейният основен учредител и главен инструктор Светлин Наков се оттегли.

Telerik Academy

През ноември 2009 г. Светлин Наков е поканен от световноизвестната софтуерната корпорация Телерик (www.telerik.com) за директор на направление "технологично обучение" и от тогава отговаря за развитие на вътрешнофирменото обучение, за организирането и провеждането на университетски курсове по съвременни софтуерни технологии и най-вече за проекта Telerik Academy (http://academy.telerik.com).

Безплатно обучение и работа по програмата "Академия на Телерик"

Програмата Telerik Academy е инициатива на иновативната и бързо растяща българска софтуерна компания Telerik за привличане на талантливи и мотивирани млади хора и задълбоченото им практическо обучение за софтуерни инженери в областта на Microsoft .NET технологиите с цел натрупване на солидни знания, сериозни практически умения и стил на мислене, необходими за започване на дългосрочна работа в Telerik. Всички обучения са безплатни, като от определен етап нататък обучаемите получават и стипендии, т. е. плаща им се, за да учат, докато работят на непълен работен ден. Курсовете в Telerik Academy следват следната програма за обучение и развитие:


Data-Centric Development
Quality Assurance and Test Automation
Windows Presentation Foundation (WPF)
ASP.NET & AJAX
C# Fundamentals (Part I)
C# Fundamentals (Part II)
.NET Essentials
Silverlight
Client Solutions Software Engineering
Windows Forms
Visual Studio Development

Курсът "C# Fundamentals" дава основните принципи на програмирането за 3 месеца и е разработен следвайки стриктно съдържанието на настоящата книга, която се ползва като основен учебник за този курс. На сайта на академията на Телерик (http://academy.telerik.com) можете да намерите видеолекции и учебни материали за всяка от главите на настоящата книга и много други полезни ресурси.

Останалите курсове в Академията за софтуерни инженери на Телерик обхващат в дълбочина съвременните .NET технологии, базите данни, уеб приложенията, настолните приложения и много други технологии. Те изграждат солидни умения за разработка на софтуер върху .NET платформата като за 4-5 месеца (целодневно) изграждат у курсиста основите на професията "софтуерен инженер". Успешно завършилите започват работа по специалността в Telerik Corporation.

Има няколко направления, за които Телерик предлага безплатни обучения и работа за успешно завършилите: разработка на софтуер с .NET технологиите (за .NET програмисти), Software Quality Assurance and Test

Automation (за QA инженери по качеството на софтуера), поддръжка на софтуера и работа с клиенти (Clients Solutions Software Engineering).

Работата във фирма Телерик (www.telerik.com) няма нужда от реклама: фирмата е работодател #1 на България (глобално, не само за ИТ индустрията) вече няколко години и това се дължи на професионализма на работа, сериозните проекти, приятната работна среда и прекрасния колектив.

Безплатни курсове по разработка на софтуер и софтуерни технологии

Освен курсовете, с които Телерик произвежда добре подготвени .NET софтуерни инженери за да попълни постоянно растящите си нужди от кадърни софтуерни специалисти, в Академията на Телерик се провеждат и редица други безплатни обучения:

- Разработка на уеб сайтове с HTML5, CSS и JavaScript (Web Front-End Development) – http://frontendcourse.telerik.com
- Разработка на уеб приложения с .NET Framework и ASP.NET – http://aspnetcourse.telerik.com
- Качествен програмен код – http://codecourse.telerik.com
- Разработка на мобилни приложения (cross-platform, iPhone, Android, Windows Phone) – http://mobiledevcourse.telerik.com

Учебните материали и видеозаписи на учебните занятия, достъпни напълно безплатно и без ангажименти, ще намерите на сайтовете на съответните курсове.

Академия на Телерик по разработка на софтуер за ученици

Безплатните обучения на Телерик не се ограничават само до изброените курсове. За всички ученици, се предлага дългосрочна безплатна подготовка за ИТ олимпиадата (НОИТ) и сериозен задълбочен курс по разработка на софтуер с .NET технологиите (и не само), който включва езика C#, бази данни, SQL и ORM технологии, уеб технологии, HTML5, ASP.NET, WPF (Windows Presentation Foundation), Silverlight, game development, mobile development, софтуерно инженерство, работа в екип и още десетки теми от разработката на софтуер. По идея обученията са за ученици и учители, но когато има свободни места в учебните зали, се допускат и външни участници.

Всички учебни материали и видеозаписи на всички проведени обучения са публикувани за свободно изтегляне и гледане от официалния сайт на Академията на Телерик за ученици: http://schoolacademy.telerik.com.

Кога и как да се запишем за безплатните обучения?

Нови безплатни курсове по разработка на софтуер и различни актуални технологии в Академията на Телерик започват на всеки няколко месеца.

Актуална информация за тях и как да се запишете ще намерите на сайта на Telerik Academy: http://academy.telerik.com.

Информация за всички предстоящи и минали безплатни курсове и обучения по разработка на софтуер и съвременни софтуерни технологии можете да получите и от личния сайт на Светлин Наков: www.nakov.com.

Поглед към съдържанието на книгата

Нека сега разгледаме накратко какво ни предстои в следващите глави на книгата. Ще разкажем по няколко изречения за всяка от тях, за да знаете какво ви очаква да научите.

Глава 1. Въведение в програмирането

В главата "Въведение в програмирането" ще разгледаме основните термини от програмирането и ще напишем първата си програма. Ще се запознаем с това какво е програмиране и каква е връзката му с компютрите и програмните езици. Накратко ще разгледаме основните етапи при писането на софтуер. Ще направим въведение в езика C# и ще се запознаем с .NET Framework и технологиите, свързани с него. Ще разгледаме какви помощни средства са ни необходими, за да можем да програмираме на C#. Ще използваме C#, за да напишем първата си програма, ще я компилираме и изпълним както от командния ред, така и от среда за разработка Microsoft Visual Studio 2010 Express Edition. Ще се запознаем още и с MSDN Library – документацията на .NET Framework, която позволява по-нататъшно изследване на възможностите на езика.

Автор на главата е Павел Дончев, а редактори са Теодор Божиков и Светлин Наков. Съдържанието на главата е донякъде базирано на работата на Лъчезар Цеков от книгата "Въведение в програмирането с Java".

Глава 2. Примитивни типове и променливи

В главата "Примитивни типове и променливи" ще разгледаме примитивните типове и променливи в C# – какво представляват и как се работи с тях. Първо ще се спрем на типовете данни – целочислени типове, реални типове с плаваща запетая, булев тип, символен тип, низов тип и обектен тип. Ще продължим с това какво е променлива, какви са нейните характеристики, как се декларира, как се присвоява стойност и какво е инициализация на променлива. Ще се запознаем и с типовете данни в C# – стойностни и референтни. Накрая ще се спрем на литералите, ще разберем какво представляват и какви литерали има.

Автори на главата са Веселин Георгиев и Светлин Наков, а редактор е Николай Василев. Съдържанието на цялата глава е базирано на работата на Христо Тодоров и Светлин Наков от книгата "Въведение в програмирането с Java".

Глава 3. Оператори и изрази

В главата "Оператори, изрази и конструкции за управление" ще се запознаем с операторите и действията, които те извършват върху различните типове данни. Ще разясним приоритетите на операторите и ще се запознаем с групите оператори според броя на аргументите, които приемат и действията, които извършват. След това ще разгледаме преобразуването на типове, защо е нужно и как да работим с него. Накрая ще опишем и покажем какво представляват изразите в C# и как се използват.

Автори на главата са Дилян Димитров и Светлин Наков, а редактор е Марин Георгиев. Съдържанието на цялата глава е базирано на работата на Лъчезар Божков от книгата "Въведение в програмирането с Java".

Глава 4. Вход и изход от конзолата

В главата "Вход и изход от конзолата" ще се запознаем с конзолата като средство за въвеждане и извеждане на данни. Ще обясним какво представлява тя, кога и как се използва, какви са принципите на повечето програмни езици за достъп до конзолата. Ще се запознаем с някои от възможностите на C# за взаимодействие с потребителя. Ще разгледаме основните потоци за входно-изходни операции **`Console.In`**, **`Console.Out`** и **`Console.Error`**, класът **`Console`** и използването на форматиращи низове за отпечатване на данни в различни формати. Ще разгледаме как можем да преобразуваме текст в число (парсване), тъй като това е начинът да въвеждаме числа в C#.

Автор на главата е Илиян Мурданлиев, а редактор е Светлин Наков. Съдържанието на цялата глава е до голяма степен базирано на работата на Борис Вълков от книгата "Въведение в програмирането с Java".

Глава 5. Условни конструкции

В главата "Условни конструкции" ще разгледаме условните конструкции в C#, чрез които можем да изпълняваме различни действия в зависимост от някакво условие. Ще обясним синтаксиса на условните оператори: **`if`** и **`if-else`** с подходящи примери и ще разясним практическото приложение на оператора за избор **`switch`**. Ще обърнем внимание на добрите практики, които е нужно да бъдат следвани, с цел постигане на по-добър стил на програмиране при използването на вложени и други видове условни конструкции.

Автор на главата е Светлин Наков, а редактор е Марин Георгиев. Съдържанието на цялата глава е базирано на работата на Марин Георгиев от книгата "Въведение в програмирането с Java".

Глава 6. Цикли

В главата "Цикли" ще разгледаме конструкциите за цикли, с които можем да изпълняваме даден фрагмент програмен код многократно. Ще разгледаме как се реализират повторения с условие (`while` и `do-while` цикли) и как се работи с `for`-цикли. Ще дадем примери за различните възможности за дефиниране на цикъл, за начина им на конструиране и за някои от основните им приложения. Накрая ще покажем как можем да използваме няколко цикъла един в друг (вложени цикли).

Автор на главата е Станислав Златинов, а редактор е Светлин Наков. Съдържанието на цялата глава е базирано на работата на Румяна Топалска от книгата "Въведение в програмирането с Java".

Глава 7. Масиви

В главата "Масиви" ще се запознаем с масивите като средства за обработка на поредица от еднакви по тип елементи. Ще обясним какво представляват масивите, как можем да декларираме, създаваме и инициализираме масиви и как можем да осъществяваме достъп до техните елементи. Ще обърнем внимание на едномерните и многомерните масиви. Ще разгледаме различни начини за обхождане на масив, четене от стандартния вход и отпечатване на стандартния изход. Ще дадем много примери за задачи, които се решават с използването на масиви и ще ви покажем колко полезни са те.

Автор на главата е Христо Германов, а редактор е Радослав Тодоров. Съдържанието на цялата глава е базирано на работата на Мариян Ненчев и Светлин Наков от книгата "Въведение в програмирането с Java".

Глава 8. Бройни системи

В главата "Бройни системи" ще разгледаме начините на работата с различни бройни системи и представянето на числата в тях. Повече внимание ще отделим на представянето на числата в десетична, двоична и шестнадесетична бройна система, тъй като те се използват много често в компютърната, комуникационната техника и в програмирането. Ще обясним и начините за кодиране на числовите данни в компютъра и видовете кодове, а именно: прав код, обратен код, допълнителен код и двоично-десетичен код.

Автор на главата е Теодор Божиков, а редактор е Михаил Стойнов. Съдържанието на цялата глава е базирано на работата на Петър Велев и Светлин Наков от книгата "Въведение в програмирането с Java".

Глава 9. Методи

В главата "Методи" ще се запознаем подробно с подпрограмите в програмирането, които в C# се наричат методи. Ще обясним кога и защо се използват методи. Ще покажем как се декларират методи и какво е сигнатура на метод. Ще научим как да създадем собствен метод и съответно как да го използваме (извикваме) в последствие. Ще демонстрираме как можем да използваме параметри в методите и как да връщаме резултат от метод. Накрая ще дискутираме някои утвърдени практики при работата с методи. Всичко това ще бъде подкрепено с подробно обяснени примери и допълнителни задачи.

Автор на главата е Йордан Павлов, а редактори са Радослав Тодоров и Николай Василев. Съдържанието на цялата глава е базирано на работата на Николай Василев от книгата "Въведение в програмирането с Java".

Глава 10. Рекурсия

В главата "Рекурсия" ще се запознаем с рекурсията и нейните приложения. Рекурсията представлява мощна техника, при която един метод извиква сам себе си. С нея могат да се решават сложни комбинаторни задачи, при които с лекота могат да бъдат изчерпвани различни комбинаторни конфигурации. Ще ви покажем много примери за правилно и неправилно използване на рекурсия и ще ви убедим колко полезна може да е тя.

Автор на главата е Радослав Иванов, а редактор е Светлин Наков. Съдържанието на цялата глава е базирано на работата на Радослав Иванов и Светлин Наков от книгата "Въведение в програмирането с Java".

Глава 11. Създаване и използване на обекти

В главата "Създаване и използване на обекти" ще се запознаем накратко с основните понятия в обектно-ориентираното програмиране – класовете и обектите – и ще обясним как да използваме класовете от стандартните библиотеки на .NET Framework. Ще се спрем на някои често използвани системни класове и ще покажем как се създават и използват техни инстанции (обекти). Ще разгледаме как можем да осъществяваме достъп до полетата на даден обект, как да извикваме конструктори и как да работим със статичните полета в класовете. Накрая ще обърнем внимание на понятието пространства от имена (namespaces), с какво те ни помагат, как да ги включваме и използваме.

Автор на главата е Теодор Стоев, а редактор е Стефан Стаев. Съдържанието на цялата глава е базирано на работата на Теодор Стоев и Стефан Стаев от книгата "Въведение в програмирането с Java".

Глава 12. Обработка на изключения

В главата "Обработка на изключения" ще се запознаем с изключенията в обектно-ориентираното програмиране, в частност в езика C#. Ще се научим как да ги прихващаме чрез конструкцията **`try-catch`**, как да ги предаваме на предходните методи и как да хвърляме собствени или прихванати изключения чрез конструкцията **`throw`**. Ще дадем редица примери за използването им. Ще разгледаме типовете изключения и йерархията, която образуват в .NET Framework. Накрая ще се запознаем с предимствата при използването на изключения и с това как най-правилно да ги прилагаме в конкретни ситуации.

Автор на главата е Михаил Стойнов, а редактор е Радослав Кирилов. Съдържанието на цялата глава е базирано на работата на Лъчезар Цеков, Михаил Стойнов и Светлин Наков от книгата "Въведение в програмирането с Java".

Глава 13. Символни низове

В главата "Символни низове" ще се запознаем със символните низове: как са реализирани те в C# и по какъв начин можем да обработваме текстово съдържание. Ще прегледаме различни методи за манипулация на текст; ще научим как да извличаме поднизове по зададени параметри, как да търсим за ключови думи, както и да отделяме един низ по разделители. Ще предоставим полезна информация за регулярните изрази и ще научим по какъв начин да извличаме данни, отговарящи на определен шаблон. Накрая ще се запознаем с методи и класове за по-елегантно и стриктно форматиране на текстовото съдържание на конзолата, с различни методики за извеждане на числа и дати.

Автор на главата е Веселин Георгиев, а редактор е Радослав Тодоров. Съдържанието на цялата глава е базирано на работата на Марио Пешев от книгата "Въведение в програмирането с Java".

Глава 14. Дефиниране на класове

В главата "Дефиниране на класове" ще покажем как можем да дефинираме собствени класове и кои са елементите на класовете. Ще се научим да декларираме полета, конструктори и свойства в класовете. Ще припомним какво е метод и ще разширим знанията си за модификатори и нива на достъп до полетата и методите на класовете. Ще разгледаме особеностите на конструкторите и подробно ще обясним как обектите се съхраняват в динамичната памет и как се инициализират полетата им. Накрая ще обясним какво представляват статичните елементи на класа – полета (включително константи), свойства и методи и как да ги ползваме.

Автори на главата са Николай Василев, Мира Бивас и Павлина Хаджиева. Съдържанието на цялата глава е базирано на работата на Николай Василев от книгата "Въведение в програмирането с Java".

Глава 15. Текстови файлове

В главата "Текстови файлове" ще се запознаем с основните похвати при работа с текстови файлове в .NET Framework. Ще разясним какво е това поток, за какво служи и как се ползва. Ще обясним какво е текстов файл и как се чете и пише в текстови файлове. Ще демонстрираме и обясним добрите практики за прихващане и обработка на изключения при работата с файлове. Разбира се, всичко това ще онагледим и демонстрираме на практика с много примери.

Автор на главата е Радослав Кирилов, а редактор е Станислав Златинов. Съдържанието на цялата глава е базирано на работата на Данаил Алексиев от книгата "Въведение в програмирането с Java".

Глава 16. Линейни структури от данни

В главата "Линейни структури от данни" ще се запознаем с някои от основните представяния на данните в програмирането и с линейните структури от данни, тъй като много често, за решаване на дадена задача се нуждаем да работим с последователност от елементи. Например, за да прочетем тази книга, трябва да прочетем последователно всяка една страница т.е. да обходим последователно всеки един от елементите на множеството от нейните страници. Ще видим как при определена задача една структура е по-ефективна и удобна от друга. Ще разгледаме структурите "списък", "стек" и "опашка" и тяхното приложение. Подробно ще се запознаем и с някои от реализациите на тези структури.

Автор на главата е Цвятко Конов, а редактор е Дилян Димитров. Съдържанието на глава е базирано в голяма степен на работата на Цвятко Конов и Светлин Наков от книгата "Въведение в програмирането с Java".

Глава 17. Дървета и графи

В главата "Дървета и графи" ще разгледаме т. нар. дървовидни структури от данни, каквито са дърветата и графите. Познаването на свойствата на тези структури е важно за съвременното програмиране. Всяка от тях се използва за моделирането на проблеми от реалността, които се решават ефективно с тяхна помощ. Ще разгледаме в детайли какво представляват дървовидните структури от данни и ще покажем техните основни предимства и недостатъци. Ще дадем примерни реализации и задачи, демонстриращи реалната им употреба. Ще се спрем по-подробно на двоичните дървета, наредените двоични дървета за претърсване и балансираните дървета. Ще разгледаме структурата от данни "граф", видовете графи и тяхната употреба. Ще покажем и къде в .NET Framework се използват имплементации на балансирани дървета за търсене.

Автор на главата е Веселин Колев, а редактор е Илиян Мурданлиев. Съдържанието на цялата глава е базирано на работата на Веселин Колев от книгата "Въведение в програмирането с Java".

Глава 18. Речници, хеш-таблици и множества

В главата "Речници, хеш-таблици и множества" ще разгледаме някои по-сложни структури от данни като речници и множества, и техните реализации с хеш-таблици и балансирани дървета. Ще обясним в детайли какво представляват хеширането и хеш-таблиците и защо са толкова важни в програмирането. Ще дискутираме понятието "колизия" и как се получават колизиите при реализация на хеш-таблици и ще предложим различни подходи за разрешаването им. Ще разгледаме абстрактната структура данни "множество" и ще обясним как може да се реализира чрез речник и чрез балансирано дърво. Ще дадем примери, които илюстрират приложението на описаните структури от данни в практиката.

Автор на главата е Михаил Вълков, а редактор е Цвятко Конов. Съдържанието на цялата глава е частично базирано на работата на Владимир Цанев от книгата "Въведение в програмирането с Java".

Глава 19. Структури от данни – съпоставка и препоръки

В главата "Структури от данни – съпоставка и препоръки" ще съпоставим една с друга структурите данни, които се разглеждат в предходните глави, по отношение на скоростта, с която извършват основните операции (добавяне, търсене, изтриване и т.н.). Ще дадем конкретни препоръки в какви ситуации какви структури от данни да ползваме. Ще обясним кога да предпочетем хеш-таблица, кога масив, кога динамичен масив, кога множество, реализирано чрез хеш-таблица и кога балансирано дърво. Всички тези структури имат вградена имплементация в .NET Framework. От нас се очаква единствено да се научим да преценяваме кога коя структура да ползваме, за да пишем ефективен и надежден програмен код.

Автори на главата са Николай Недялков и Светлин Наков, а редактор е Веселин Колев. Съдържанието на цялата глава е базирано на работата на Светлин Наков и Николай Недялков от книгата "Въведение в програмирането с Java".

Глава 20. Принципи на обектно-ориентираното програмиране

В главата "Принципи на обектно-ориентираното програмиране" ще се запознаем с принципите на обектно-ориентираното програмиране: наследяване на класове и имплементиране на интерфейси, абстракция на данните и на поведението, капсулация на данните и скриване на информация за имплементацията на класовете, полиморфизъм и виртуални методи. Ще обясним в детайли принципите за свързаност на отговорностите и функционално обвързване (cohesion и coupling). Ще опишем накратко как се извършва обектно-ориентирано моделиране и как се създава обектен модел по описание на даден бизнес проблем. Ще се запознаем с езика UML и ролята му в процеса на обектно-ориентираното моделиране. Накрая ще разгледаме съвсем накратко концепцията "шаблони за дизайн" и ще дадем няколко типични примера за шаблони, широко използвани в практиката.

Автор на главата е Михаил Стойнов, а редактор е Михаил Вълков. Съдържанието на цялата глава е базирано на работата на Михаил Стойнов от книгата "Въведение в програмирането с Java".

Глава 21. Качествен програмен код

В главата "Качествен програмен код" ще разгледаме основните правила за писане на качествен програмен код. Ще бъде обърнато внимание на именуването на елементите от програмата (променливи, методи, класове и други), правилата за форматиране и подреждане на кода, добрите практики за изграждане на висококачествени класове и методи и принципите за качествена документация на кода. Ще бъдат дадени много примери за качествен и некачествен код. В процеса на работа ще бъде обяснено как да се използва средата за програмиране, за да се автоматизират някои операции като форматиране и преработка на съществуващ код, когато се налага.

Автор на главата е Михаил Стойнов, а редактор е Павел Дончев. Съдържанието на цялата глава е базирано частично на работата на Михаил Стойнов, Светлин Наков и Николай Василев от книгата "Въведение в програмирането с Java".

Глава 22. Ламбда изрази и LINQ заявки

В главата "Ламбда изрази и LINQ заявки" ще се запознаем с част от по-сложните възможности на езика C# и по-специално ще разгледаме как се правят заявки към колекции чрез ламбда изрази и LINQ заявки. Ще обясним как да добавяме функционалност към съществуващи вече класове, използвайки разширяващи методи (extension methods). Ще се запознаем с анонимните типове (anonymous types), ще опишем накратко какво представляват и как се използват. Ще разгледаме ламбда изразите (lambda expressions) и ще покажем с примери как работят повечето вградени ламбда функции. След това ще обърнем по-голямо внимание на езика за заявки LINQ, който е част от C#. Ще научим какво представлява, как работи и какви заявки можем да конструираме с него. Накрая ще се запознаем с ключовите думи за езика LINQ, тяхното значение и ще ги демонстрираме, чрез голям брой примери.

Автор на главата е Николай Костов, а редактор е Веселин Колев.

Глава 23. Как да решаваме задачи по програмиране?

В главата "Как да решаваме задачи по програмиране?" ще дискутираме един препоръчителен подход за решаване на задачи по програмиране и ще го илюстрираме нагледно с реални примери. Ще дискутираме инженерните принципи, които трябва да следваме при решаването на задачи (които важат в голяма степен и за задачи по математика, физика и други дисциплини) и ще ги покажем в действие. Ще опишем стъпките, през които преминаваме при решаването на няколко примерни задачи и ще демонстрираме какви грешки се получават, ако не следваме тези стъпки. Ще обърнем внимание на някои важни стъпки от решаването на задачи (като например тестване), които обикновено се пропускат.

Автор на главата е Светлин Наков, а редактор е Веселин Георгиев. Съдържанието на цялата глава е базирано изцяло на работата на Светлин Наков от книгата "Въведение в програмирането с Java".

Глави 24, 25, 26. Практически задачи за изпит по програмиране

В главите "Практически задачи за изпит по програмиране" ще разгледаме условията и ще предложим решения на девет примерни задачи от три примерни изпита по програмиране. При решаването им ще приложим на практика описаната методология в главата "Как да решаваме задачи по програмиране".

Автори на главите са съответно Стефан Стаев, Йосиф Йосифов и Теодор Стоев, а редактори са съответно Радослав Тодоров, Радослав Иванов и Йосиф Йосифов. Съдържанието на тези глави е базирано в голяма степен на работата на Стефан Стаев, Светлин Наков, Радослав Иванов и Теодор Стоев от книгата "Въведение в програмирането с Java".

За използваната терминология

Тъй като настоящият текст е на български език, ще се опитаме да ограничим употребата на английски термини, доколкото е възможно. Съществуват обаче основателни причини да използваме и английските термини наред с българските им еквиваленти:

- По-голямата част от техническата документация за C# и .NET Framework е на английски език (повечето книги и в частност официалната документация) и затова е много важно читателите да знаят английския еквивалент на всеки използван термин.
- Много от използваните термини не са пряко свързани със C# и са навлезли отдавна в програмисткия жаргон от английски език (например "дебъгвам", "компилирам" и "плъгин"). Тези термини ще бъдат изписвани най-често на кирилица.
- Някои термини (например "framework" и "deployment") са трудно преводими и трябва да се използват заедно с оригинала в скобки. В настоящата книга на места такива термини са превеждани по различни начини (според контекста), но винаги при първо срещане се дава и оригиналният термин на английски език.

Как възникна тази книга?

Често се случва някой да попита от коя книга да започне да се учи на програмиране. Срещат се ентусиазирани младежи, които искат да се учат да програмират, но не знаят от къде да започнат. За съжаление е трудно да им бъде препоръчана добра книга. Можем да се сетим за много книги за C# – и на български и на английски, но никоя от тях не учи на програмиране. Няма много книги (особено на български език), които да учат на концепциите на програмирането, на алгоритмично мислене, на структури от данни. Има книги за начинаещи, които учат на езика C#, но не и на основите на програмирането. Има и няколко хубави книги за програмиране на български език, но са вече остарели и учат на отпаднали при еволюцията езици и технологии. Известни са няколко такива книги за C и Паскал, но не и за C# или Java. В крайна сметка е трудно да се сетим за хубава книга, която горещо да препоръчаме на всеки, който иска да се захване с програмиране от нулата.

Липсата на хубава книга по програмиране за начинаещи в един момент мотивира главния организатор на проекта Светлин Наков да събере авторски колектив, който да се захване и да напише най-накрая такава книга. Решихме, че можем да помогнем и да дадем знания и вдъхновение на много млади хора да се захванат сериозно с програмиране.

Историята на тази книга

Тази книга възникна като превод и адаптация на книгата "Въведение в програмирането с Java" към C# и .NET Framework и съответно наследява историята на своя предшественик като добавя нови нотки на нововъведения, изменения и допълнения от новия авторски колектив.

Историята на книгата "Въведение в програмирането с Java" е дълга и интересна. Тя започва с въведителните курсовете по програмиране в Национална академия по разработка на софтуер (НАРС) през 2005 г., когато под ръководството на Светлин Наков за тях е изготвено учебно съдържание за курс "Въведение в програмирането със C#". След това то е адаптирано към Java и така се получава курсът "Въведение в програмирането с Java". През годините това учебно съдържание претърпява доста промени и подобрения и достига до един изчистен и завършен вид.

Събиране на авторския екип

Работата по оригиналната книга "Въведение в програмирането с Java" започва в един топъл летен ден (август 2008 г.), когато основният автор Светлин Наков, вдъхновен от идеята за написване на учебник за курсовете по "Въведение в програмирането" събира екип от двадесетина млади софтуерни инженери, ентусиасти, които имат желание да споделят знанията си и да напишат по една глава от книгата.

Светлин Наков дефинира учебното съдържание и го разделя в глави и създава шаблон за съдържанието на всяка глава. Шаблонът съдържа структурата на текста – всички основни заглавия в дадената глава и всички подзаглавия. Остава да се напишат текста, примерите и задачите.

На първата среща на екипа учебното съдържание претърпява малко промени. По-обемните глави се разделят на няколко отделни части (например структурите от данни), възникват няколко нови глави (например работа с изключения) и се определят автори и редактори за всяка глава. Идеята е проста: всеки да напише по 1 глава от книгата и накрая да бъдат съединени в книга. За да няма голяма разлика в стиловете, форматирането и начина на представяне на информацията авторите приемат единно ръководство на писателя, в което строго се описват всички правила за писане. В крайна сметка всеки си има тема и писането започва.

За проекта се създава сайт за съвместна работа в екип в Google Code на адрес http://code.google.com/p/introjavabook/, където стои последната версия на всички текстове и материали по книгата.

Задачите и сроковете

Както във всеки проект, след разпределяне на задачите се слагат крайни срокове за всяка от тях, за да се планира работата във времето. По план книгата трябва да излезе от печат през октомври 2008 г., но това не се случва в срок, защото много от авторите се забавят, а някои въобще не изпълняват поетия ангажимент.

Когато идва първия краен срок едва половината от авторите са готови на време. Сроковете се удължават и голяма част от авторите завършват работата по своята глава. Започва работата на редакторите. Паралелно някои автори дописват. За някои глави се търсят нови автори, защото оригиналният автор се проваля и бива отстранен.

Няколко месеца по-късно книгата е готова на 90%, авторите загубват ентусиазъм работата започва да върви много бавно и мъчно. Светлин Наков се опитва да компенсира и да дописва недовършените теми, но работата е много. Въпреки 30-те часа, които той влага като труд всеки свободен уикенд, работата е много и все не свършва месеци наред.

Всички автори подценяват сериозно обема на работата и това е основно причината за забавянето на нейната поява. Авторите си мислят, че писането става бързо, но истината е, че за една страница текст (четене, писане, редактиране, преправяне и т.н.) отива средно по 1 час работа, та дори и повече. Сумарно за написването на цялата книга са вложени около 800-1000 работни часа труд, разпределени сред всички автори и редактори, което се равнява на над 6 месеца работа на един автор на пълен работен ден. Понеже всички автори пишат в свободното си време, работата върви бавно и отнема 4-5 месеца.

Книгата "Въведение в програмирането с Java" излиза официално през януари 2009 г. и се разпространява безплатно от официалния й уеб сайт: www.introprogramming.info.

Превеждане на книгата към C#

Книгата "Въведение в програмирането с Java" се чете с голям интерес. Към декември 2009 г. тя е изтеглена над 6 000 пъти и първият тираж на хартия е почти изчерпан, а сайтът й е посетен над 50 пъти на ден.

През ноември 2009 г. стартира проект за "превеждане" на книгата "Въведение в програмирането с Java" към C# под заглавие "Въведение в програмирането със C#". Събира се отново голям екип от софтуерни инженери под ръководството на Светлин Наков и Веселин Колев. Идеята е да се адаптира текста на книгата, заедно с всички примери, демонстрации, обяснения към тях, задачи, решения, упражнения и упътвания към C# и .NET Framework. Работата изглежда, че не е много – трябва да се прочете внимателно текста, да се адаптира за C#, да се преработят всички примери и да се заменят всички класове, методи и технологии, свързани с Java със съответните им C# класове, методи и технологии. Лесна на пръв поглед задача, която обаче се оказва времеотнемаща. Както може да се очаква, при проекти, които се разработват от широк колектив автори, в свободно им време и на добра воля, книгата е завършена за около половин година. Тогава излиза предварителната версия на книгата, в която са открити доста грешки и неточности. За да бъдат изчистени, екипът работи още около година и успява да изглади текста, примерите и задачите за упражнения до вид, подходящ за официално издаване на хартия. Някои от главите се налага да бъдат сериозно редактирани, почти пренаписани, добавя се и главата за ламбда изрази и LINQ.

Новият проект също е с отворен код и работата по него е публично достъпна в Google Code: http://code.google.com/p/introcsharpbook/. Книгата остава със същия брой глави и няма сериозни промени по същество. За автори и редактори са поканени всички оригинални автори на съответните глави от книгата "Въведение в програмирането с Java", но повечето от тях се отказват и към екипа се присъединяват много нови автори. В крайна сметка проектът завършва с успех и книгата "Въведение в програмирането със C#" излиза през лятото на 2011 г. Сайтът на новата книга е същият (www.introprogramming.info), като е разделен на секция за C# и Java.

Част от авторите проявяват интерес за адаптиране на книгата още веднъж, към езика C++, но не е твърдо решено ще бъде ли стартиран такъв проект и евентуално кога. Има и идеи за превод на книгата на английски език, но за такава амбициозна задача е необходим сериозен екип и много труд, както и инициативен и усърден ръководител. Надяваме някой ден и двата проекта да се случат, но нищо не обещаваме.

Авторският колектив

Авторският колектив (на старата и на новата книга) е наистина главният виновник за съществуването на тази книга. Написването на текст с такъв обем и такова качество е сериозна задача, която изисква много време.

Идеята за участие на толкова много автори е добре проверена, тъй като по подобен начин са написани вече няколко други книги (като "Програмиране за .NET Framework" – част 1 и 2). Въпреки, че отделните глави от книгата са писани от различни автори, те следват единен стил и високо качество (макар и не еднакво във всички глави). Текстът е добре структуриран, с много заглавия и подзаглавия, с много и подходящи примери, с добър стил на изказ и еднакво форматиране.

Екипът, написал настоящата книга, е съставен от хора, които имат силен интерес към програмирането и желаят безвъзмездно да споделят своите знания като участват в написването на една или няколко от главите. Най-хубавото е, че всички автори, съавтори и редактори от екипа по разработката на книгата са действащи програмисти с реален практически опит, което означава, че читателят ще почерпи знания, практики и съвети от хора, реализирали се в софтуерната индустрия.

Участниците в проекта дадоха своя труд безвъзмездно, без да получат материални или други директни облаги, защото подкрепяха идеята за написване на добра книга за начинаещи програмисти на български език и имаха силно желание да помогнат на своите бъдещи колеги да навлязат бързо в програмирането.

Следва кратко представяне на авторите на книгата "Въведение в програмирането със C#" (по азбучен ред). Оригиналните автори на съответните глави от книгата "Въведение в програмирането с Java" също са упоменати по подходящ начин, тъй като техните заслуги в някои глави са по-големи, отколкото заслугите на следващите автори след тях, които са адаптирали текста и примерите към C#.

Веселин Георгиев

Веселин Георгиев е съосновател на Lead IT (www.leadittraining.com) и софтуерен разработчик в Abilitics (www.abilitics.com). В момента е магистър "Електронен бизнес и Електронно управление" в Софийски Университет "Св. Климент Охридски", след завършена бакалавърска степен по Информатика също в Софийски Университет.

Веселин е Microsoft Certified Trainer. Бил е лектор в конференциите "Дни на Майкрософт" през 2011 и 2009 г.. Участва като преподавател в курсовете "Програмиране с .NET & WPF" и "Разработка на богати интернет приложения (RIA) със Silverlight" в Софийски Университет. Опитен лектор, работил върху обучението на софтуерни специалисти за практическа работа в ИТ индустрията.

Професионалните му интереси са насочени към .NET обучения, разработката на разнообразни .NET приложения, софтуерни архитектури. Сертифициран е като Microsoft Certified Professional Developer.

Личният технологичен блог на Веселин Георгиев е достъпен от адрес http://veselingeorgiev.net/blog/. Можете да се свържете с него по e-mail: veselin.vgeorgiev@gmail.com.

Веселин Колев

Веселин (Веско) Колев е водещ софтуерен инженер с дългогодишен професионален опит. Той е работил с различни компании, в които е ръководил разработката на разнообразни софтуерни проекти и екипи. Като ученик е участвал в редица състезания по математика, информатика и информационни технологии, в които е заемал престижни места. В момента следва специалност Компютърни науки във Факултета по математика и информатика на СУ "Св. Климент Охридски".

Веско е опитен лектор, работил върху обучението на софтуерни специалисти за практическа работа в ИТ индустрията. Участва като преподавател във Факултета по математика и информатика на Софийски университет, където до сега е водил курсовете "Съвременни Java технологии" и "Качествен програмен код". Водил е аналогични лекции и в Технически университет, София.

Основните интереси на Веско включват дизайн на софтуерни проекти, разработване на софтуерни системи, .NET и Java технологиите, програмиране с Win32 (C/C++), софтуерни архитектури, шаблони за дизайн, алгоритми, бази от данни, управление на екипи и проекти за разработване на софтуер, обучение на специалисти. Проектите, по които е работил, включват големи уеб базирани системи, мобилни приложения, OCR, системи за машинен превод, икономически софтуер и много други. Веско е съавтор и в книгата "Въведение в програмирането с Java".

В момента, Веско се специализира в разработката на приложения базирани на Silverlight и WPF във фирма Телерик (www.telerik.com). Част от своя ежедневен опит, той споделя онлайн в личния си блог, достъпен от адрес http://veskokolev.blogspot.com.

Дилян Димитров

Дилян Димитров е сертифициран софтуерен разработчик с професионален опит в изграждането на средни и големи уеб базирани системи върху .NET платформата. Интересите му включват разработка, както на уеб, така и на десктоп приложения с последните технологии на Microsoft. Той е завършил Факултета по математика и информатика на Софийския университет "Св. Климент Охридски" със специалност "Информатика". Може да се свържете с него по e-mail: dimitrov.dilqn@gmail.com или да посетите личният му блог на адрес: http://dilyandimitrov.blogspot.com.

Илиян Мурданлиев

Илиян Мурданлиев е софтуерен разработчик във фирма Ниърсофт (www.nearsoft.eu). В момента е магистър "Компютърни Технологии и Приложно Програмиране" в Технически Университет - София. Бакалавър е в същия университет в специалност "Приложна Математика". Завършил е езикова гимназия с английски език.

Илиян е участвал в сериозни проекти и е участвал при разработката както на front-end визуализацията, така и на back-end логиката. Съставял е и е водил лекции по C# и други езици за програмиране. Интересите на Илиян са в областта на новите технологии свързани с .NET, Windows Forms и

Web базираните технологии, шаблони за дизайн, алгоритми и софтуерно инженерство. Обича разчупени проекти, в които не трябват само познания, но и логическо мислене.

Личният му блог е достъпен на адрес: http://imurdanliev.wordpress.com. Можете да се свържете с него по e-mail: i.murdanliev@gmail.com.

Йосиф Йосифов

Йосиф Йосифов е софтуерен разработчик в Telerik (www.telerik.com). Интересите му са свързани предимно с .NET технологиите, шаблоните за дизайн и компютърните алгоритми. Участвал е в множество състезания и олимпиади по информатика. В момента той следва Компютърни науки във Факултета по математика и информатика на Софийски Университет "Св. Климент Охридски".

Личният блог на Йосиф е достъпен от адрес: http://yyosifov.blogspot.com. Можете да се свържете с него по e-mail: cypressx@gmail.com.

Йордан Павлов

Йордан Павлов е завършил бакалавърска и магистърска степен, специалност "Компютърни системи и технологии" в Технически университет – София. Той е софтуерен разработчик в Телерик (www.telerik.com) със значителен опит в разработката на софтуерни компоненти.

Интересите му са най-вече в следните области: обектно-ориентиран дизайн, шаблони за дизайн, разработка на качествен софтуер, географски информационни системи (ГИС), паралелна обработка и високо производителни системи, изкуствен интелект, управление на екипи.

Йордан е победител на локалните финали за България на състезанието Imagine Cup 2008 в категория "Софтуерен дизайн" както и на световните финали в Париж, където печели престижната награда на Microsoft - "The Engineering Excellence Achievement Award". Работил е заедно с инженери на Майкрософт в централата на компанията в Редмънд, САЩ, където е натрупал полезни знания и умения за разработката на сложни софтуерни системи.

Йордан е и носител на златен знак за "принос към младежкото иновационно и информационно общество". Участвал е в множество състезания и олимпиади по информатика.

Личният му блог му е достъпен на адрес http://yordanpavlov.blogspot.com. Можете да се свържете с него по e-mail: iordanpavlov@gmail.com.

Мира Бивас

Мира Бивас е ентусиазиран млад програмист в един от ASP.NET екипите на **t**elerik (www.telerik.com). Тя е студентка във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски", специалност "Приложна математика". Мира е завършила Intro C# и Core .NET курсовете на Национална академия по разработка на софтуер (НАРС).

Може да се свържете с нея на e-mail: mira.bivas@gmail.com.

Михаил Вълков

Михаил Вълков е софтуерен разработчик от 2000 г. През годините Михаил се е сблъсквал с множество технологии и платформи за разработка, сред които MS .NET, ASP, Delphi. От 2004 г. Михаил разработва софтуер във фирма Телерик (www.telerik.com). Там той участва в изграждането на редица компоненти за ASP.NET, Windows Forms, Silverlight и WPF. През последните години, Михаил ръководи едни от най-добре развиващите се екипи в компанията.

Можете да се свържете с него чрез e-mail: m.valkov@gmail.com.

Неговият блог е достъпен от: http://blogs.telerik.com/mihailvalkov/.

Михаил Стойнов

Михаил Стойнов е магистър по "Стопанско управление" в Софийски университет. Бакалавърската си степен по "Информатика" е завършил отново в Софийски Университет. Понастоящем е ръководител на развойната дейност в Матерна България (www.materna.bg).

Михаил е професионален разработчик на софтуер, консултант и преподавател с дългогодишен опит. От няколко години той е хоноруван преподавател във Факултета по математика и информатика като досега е водил лекции в курсовете "Теория на мрежите", "Програмиране за .NET Framework", "Разработка на Java уеб приложения", "Шаблони за дизайн" и "Качествен програмен код". Преподавал е и в Нов български университет.

Той е автор на редица статии и публикации и лектор на множество конференции и семинари в областта на софтуерните технологии и информационната сигурност. Михаил е участвал като съавтор в книгите "Програмиране за .NET Framework" и "Въведение в програмирането с Java". Участвал е в академичната програма на Microsoft - MSDN Academic Alliance и е бил лектор в академичните дни на Майкрософт.

Михаил е водил IT обучения в България и в чужбина. Бил е лектор на курсове по Java, Java EE, SOA, Spring в Национална академия по разработка на софтуер (НАРС). Член е на Българската асоциация на разработчиците на софтуер (БАРС).

Михаил е работил в международните офиси на Siemens, HP, EDS в Холандия и Германия, където е натрупал сериозен опит както за софтуерното изкуство, така и за качественото писане на софтуер чрез участието си в големи софтуерни проекти. Неговите интереси обхващат изграждането на софтуерни архитектури и дизайн, B2B интегриране на разнородни информационни системи, оптимизация на бизнес процеси и софтуерни системи основно върху платформите Java и .NET. Михаил е участвал в десетки проекти и има значителен опит с уеб приложения и уеб услуги, разпределени системи, релационни бази от данни и ORM инструменти, и управлението на проекти и екипи за разработка на софтуер.

Личният му блог е достъпен на адрес: http://mihail.stoynov.com/blog/.

Николай Василев

Николай Василев е завършил бакалавърската си степен във Факултета по математика и информатика на Софийски университет "Св. Кл. Охридски", специалност "Математика и информатика". Има магистърска степен от университета в Малага, Испания, специалност "Софтуерно инженерство и изкуствен интелект". В момента завършва магистърската програма на Софийски университет "Св. Кл. Охридски", специалност "Уравнения на математическата физика и приложения".

Той е професионален разработчик на софтуер, като е работил, както в български, така и международни компании.

Съавтор е на книгата "Въведение в програмирането с Java".

В периода 2002-2005 г е водил упражненията към курсовете по програмиране водени от доц. Божидар Сендов, "Увод в програмирането" и "Структури от данни и програмиране".

Интересите му са свързани, както със софтуерната индустрия – проектиране, имплементация и интеграция на софтуерни системи (предимно върху платформата Java), така и с участие в академични и научно-изследователски дейности в областите на софтуерното инженерство, изкуствения интелект и механиката на флуидите.

Участвал е в множество разнородни проекти и има опит в разработката на уеб приложения и уеб услуги, релационни бази от данни и ORM платформи, модулно програмиране с OSGI, потребителски интерфейси, разпределени и VOD системи.

Личният блог на Николай Василев е на адрес: http://blog.nvasilev.com

Николай Костов

Николай Костов работи като technical trainer в отдел "технологично обучение" във фирма Телерик. Занимава се с обученията в академията на Телерик (http://academy.telerik.com) и курсовете, организирани от Телерик. Учи във Факултета по математика и информатика на Софийския университет "Св. Климент Охридски", специалност "Компютърни науки".

Николай е дългогодишен участник в редица ученически и студентски олимпиади и състезания по информатика. Двукратен победител в проектните категории "Приложни програми" и "Интернет приложения" на националната олимпиадата по информационни технологии. Има богат опит в проектирането и изграждането на интернет приложения, алгоритмичното програмиране и обработката на големи обеми данни.

Основните му интереси са свързани с разработването на софтуерни приложения, алгоритми, структури от данни, всичко свързано с .NET технологиите, сигурност на уеб приложенията, автоматизиране на обработката на данни, web crawlers и др.

Личният блог на Николай е достъпен на адрес: http://nikolay.it/.

Николай Недялков

Николай Недялков е президент на Асоциация за информационна сигурност, технически директор на портала за електронни разплащания и услуги eBG.bg и бизнес консултант в други компании. Николай е професионален разработчик на софтуер, консултант и преподавател с дългогодишен опит. Той е автор на редица статии и публикации и лектор на множество конференции и семинари в областта на софтуерните технологии и информационната сигурност. Преподавателският му опит се простира от асистент по "Структури от данни в програмирането", "Обектно-ориентирано програмиране със C++" и "Visual C++" до лектор в курсовете "Мрежова сигурност", "Сигурен програмен код", "Интернет програмиране с Java", "Конструиране на качествен програмен код", "Програмиране за платформа .NET" и "Разработка на приложения с Java". Интересите на Николай са концентрирани върху изграждането и управлението на информационни и комуникационни решения, моделирането и управлението на бизнес процеси в големи организации и в държавната администрация. Николай има бакалавърска и магистърска степен от Факултета по математика и информатика на Софийски университет "Св. Климент Охридски". Като ученик е дългогодишен състезател по програмиране, с редица призови отличия.

Личният му уеб сайт е достъпен от адрес: http://www.nedyalkov.com.

Павел Дончев

Павел Дончев е програмист във фирма **t**elerik (www.telerik.com), където се занимава с разработка на уеб приложения, предимно за вътрешни нужди на фирмата. Следва задочно теоретична физика в Софийски университет "Св. Климент Охридски". Занимавал се е с разработка на Windows и Web приложения в различни сектори на бизнеса – ипотечни кредити, онлайн магазини, автоматика, Web UML диаграми. Интересите му са предимно в сферата на автоматизирането на процеси с технологиите на Майкрософт.

Личният му блог е достъпен от адрес http://donchevp.blogspot.com.

Павлина Хаджиева

Павлина Хаджиева е програмист във фирма Телерик (www.telerik.com). В момента е магистър "Разпределени системи и мобилни технологии" във Факултета по математика и информатика на Софийски Университет "Св. Климент Охридски". Бакалавърската си степен по "Химия и Информатика" е завършила също в Софийски Университет.

Професионалните ѝ интереси са насочени към уеб технологиите, в частност ASP.NET, както и цялостната разработка на приложения, базирани на .NET Framework.

Можете да се свържете с Павлина Хаджиева на нейния e-mail: pavlina.hadjieva@gmail.com

Радослав Иванов

Радослав Иванов е софтуерен инженер, работил с широк набор от технологии. Завършил е Факултета по математика и информатика на Софийски университет "Св. Климент Охридски" и има сериозен професионален опит в разработката на софтуер. Той е лектор в редица курсове в Софийски университет "Св. Климент Охридски", частни компании и организации и е съавтор на книгите "Програмиране за .NET Framework" и "Въведение в програмирането с Java". Работил е като софтуерен инженер в Европейската организация за ядрени изследвания (CERN) – www.cern.ch. Сред професионалните му интереси са Java технологиите, .NET платформата, архитектура и дизайн на софтуерни системи и др.

Радослав Кирилов

Радослав Кирилов е софтуерен разработчик във фирма Телерик (www.telerik.com). Завършил е Технически университет – София, специалност "Компютърни системи и технологии". Професионалните му интереси са насочени към уеб технологиите, в частност ASP.NET, както и цялостната разработка на приложения, базирани на .NET Framework. Радослав е опитен лектор, участвал както в провеждането, така и в разработката на материали (презентации, примери, упражнения) за множество курсове в Национална академия по разработка на софтуер (НАРС). Радослав участва в преподавателския екип на курса "Качествен програмен код", който стартира в началото на 2010 година в Технически университет – София и в Софийски университет "Св. Климент Охридски".

Неговият технологичен блог, който той поддържа от началото на 2009 година, е достъпен на адрес http://radoslavkirilov.blogspot.com/. Можете да се свържете с Радослав на e-mail: radoslav.pkirilov@gmail.com.

Радослав Тодоров

Радослав Тодоров е софтуерен разработчик завършил бакалавърската си степен във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски" (www.fmi.uni-sofia.bg). Магистърското си образование в областта на компютърните науки получава в Датския технически университет в Люнгбю, Дания (http://www.dtu.dk).

Радослав преподава още като асистент-преподавател в курсове на IT University, Копенхаген, Дания (http://www.itu.dk) и участва в изследователска дейност в проекти на университета от магистърското си образование. Той има богат опит в проектирането, разработването и поддръжката на големи софтуерни продукти за различни компании. Трудовия му опит протича в няколко фирми в България. Към настоящия момент работи като софтуерен инженер за Canon Handy Terminal Solutions Europe в Дания (www.canon-europe.com/Handy_Terminal_Solutions/).

Интересите на Радослав са насочени както към софтуерните технологии с езици от високо ниво, така и към продукти, интегриращи цялостни хардуерни и софтуерни решения в индустриалния и частния сектор.

Може да се свържете с Радослав по e-mail: radoslav_todorov@hotmail.com.

Светлин Наков

Светлин Наков е ръководител на направление "технологично обучение" в Telerik Corporation, където ръководи проектите за безплатно обучение на софтуерни инженери Telerik Academy (http://academy.telerik.com) и Telerik School Academy (http://schoolacademy.telerik.com) и е основен преподавател в учебното звено на Telerik.

Той е завършил бакалавърска степен по информатика и магистърска степен по разпределени системи и мобилни технологии в Софийски университет "Св. Климент Охридски". По-късно получава и докторска степен (PhD) по компютърни науки с дисертация в областта на изчислителната лингвистика, защитена пред Висшата атестационна комисия (ВАК) към Българската академия на науките (БАН).

Неговите интереси обхващат изграждането на софтуерни архитектури, .NET платформата, уеб приложенията, базите данни, Java технологиите, обучението на софтуерни специалисти, информационната сигурност, технологичното предприемачество и управлението на проекти и екипи за разработка на софтуер.

Светлин Наков има 15-годишен опит като софтуерен инженер, програмист, преподавател и консултант, преминал от Assembler, Basic и Pascal през C и C++ до PHP, JavaScript, Java и C#. Участвал е като софтуерен инженер, консултант и ръководител на екипи в десетки проекти за изграждане на информационни системи, уеб приложения, системи за управление на бази от данни, бизнес приложения, ERP системи, криптографски модули и обучения на софтуерни инженери. На 24 години създава първата си софтуерна фирма за обучение на софтуерни инженери, която 5 години по-късно бива погълната от Телерик.

Светлин има сериозен опит в изграждането на учебни материали, подготовката и провеждането на курсове за обучения по програмиране и съвременни софтуерни технологии, натрупан по време на преподавателската му практика. Години наред той е хоноруван преподавател по съвременни софтуерни технологии във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски" (ФМИ на СУ), Нов Български университет (НБУ) и Технически университет – София (ТУ-София), където води курсове по "Проектиране и анализ на компютърни алгоритми", "Интернет програмиране с Java", "Мрежова сигурност", "Програмиране за .NET Framework", "Разработка на Java уеб приложения", "Шаблони за дизайн", "Качествен програмен код", "Разработка на уеб приложения с .NET Framework и ASP.NET", "Разработка на Java и Java EE приложения" и "Web Front-End Development" (вж. http://www.nakov.com/courses/).

Светлин има десетки научни и технически публикации, свързани с разработката на софтуер, в български и чуждестранни издания и е водещ автор на книгите "Програмиране за .NET Framework (том 1 и 2)", "Въведение в програмирането с Java", "Въведение в програмирането със C#", "Интернет програмиране с Java" и "Java за цифрово подписване на документи в уеб". Той е редовен лектор на технически конференции, обучения и семинари и до момента е изнесъл над 100 технически лекции по различни технологични събития в България и чужбина.

Като ученик и студент Светлин е победител в десетки национални състезания по програмиране и е носител на 4 медала от международни олимпиади по информатика.

През 2003 г. той е носител на наградата "Джон Атанасов" на фондация Еврика. През 2004 г. получава награда "Джон Атанасов" от президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество.

Той е един от учредителите на Българската асоциация на разработчиците на софтуер (www.devbg.org) и понастоящем неин председател.

Неговият личен уеб сайт и блог е достъпен от: http://www.nakov.com.

Станислав Златинов

Станислав Златинов е софтуерен разработчик с професионален опит в разработването на уеб и десктоп приложения, базирани на .NET и Java платформите.

Завършил е магистратура по Компютърна мултимедия във Великотърновски университет "Св. Св. Кирил и Методий".

Неговият личен блог е достъпен от: http://encryptedshadow.blogspot.com.

Стефан Стаев

Стефан Стаев е софтуерен разработчик, който се занимава с изграждане на уеб базирани системи на .NET платформата. Професионалните му интереси са свързани с последните .NET технологии, шаблони за дизайн и база от данни. Участник е в авторския екип на книгата "Въведение в програмирането в Java".

В момента Стефан следва специалност "Информатика" във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски". Завършил е "Националната академия по разработка на софтуер" по специалност "Core .NET Developer".

Можете да се свържете с него по e-mail: stefosv@gmail.com. Неговият Twitter микроблог е на адрес: http://twitter.com/stefanstaev.

Теодор Божиков

Теодор Божиков е софтуерен разработчик във фирма Телерик (www.telerik.com). Завършва магистратурата си по Компютърни системи и технологии в Технически университет – Варна. Освен опита си като програмист в областта на WPF и Silverlight, той е натрупал експертиза и в разработката на ASP.NET уеб приложения. За кратко се занимава с разработката на частни сайтове. В рамките на проекта i-центрове е участвал в изграждането и поддържането на локална мрежа за публично ползване във Фестивалния и конгресен център - Варна. Водил курсове по компютърна грамотност и основи на компютърните мрежи.

Професионалните интереси на Теодор включват технологии за разработка на уеб и десктоп приложения, архитектури и шаблони за дизайн, мрежи и всякакви нови технологии.

Можете да се свържете с Теодор по e-mail: t_bozhikov@yahoo.com. Неговият микроблог в Twitter е достъпен от: http://twitter.com/tbozhikov.

Теодор Стоев

Теодор Стоев е завършил бакалавърска и магистърска степен по специалност Информатика във ФМИ на Софийски университет "Св. Климент Охридски". Магистърската му специализация в СУ е "Софтуерни технологии". В момента следва магистърска програма "Computer Science" в Saarland University (Саарбрюкен, Германия).

Теодор е проектант и разработчик на софтуер с дългогодишен опит. Участвал е в изграждането на финансови и застрахователни софтуерни системи, редица уеб приложения и корпоративни сайтове. Участвал е активно в разработката на проекта TENCompetence на Европейската комисия. Съавтор е на книгата "Въведение в програмирането с Java".

Неговите професионални интереси са в областта на обектно-ориентирания анализ, моделиране и изграждане на софтуерни приложения, уеб технологиите и в частност изграждането на RIA (Rich Internet Applications). Зад гърба си има немалък опит с алгоритмично програмиране: участвал е в редица ученически и студентски национални състезания по информатика.

Неговият личен сайт е достъпен от адрес: http://www.teodorstoev.com.

Можете да се свържете с Теодор по e-mail: teodor.stoev@gmail.com.

Христо Германов

Христо Германов е софтуерен инженер, чиито интереси са свързани предимно с .NET технологиите. Архитектурата и дизайна на уеб базирани системи, алгоритмите и съвременните стандарти за качествен код са също негова страст. Участвал е в разработката както на малки, така и на големи

Web и Desktop базирани приложения. Обича предизвикателни задачи и проекти, в които се изисква силно логическо мислене. Завършил е специалност "Компютърни мрежи" в колеж "Омега" гр. Пловдив и е специализирал в "Националната академия по разработка на софтуер", София, по специалност "Core .NET Developer".

Можете да се свържете с него по e-mail: hristo.germanov@gmail.com.

Цвятко Конов

Цвятко Конов е софтуерен разработчик и преподавател с разностранни интереси и опит. В неговите компетенции влизат области като интеграции на системи, изграждане на софтуерни архитектури, разработване на системи с редица технологии като .NET 4.0, ASP.NET, Silverlight, WPF, WCF, RIA, MS SQL Server, Oracle, MySQL, PostgreSQL и PHP. Преподавателския му опит включва голяма палитра от курсове – курсове за начинаещи и напреднали върху .NET технологиите, както и специализирани курсове в отделни технологии като ASP.NET, Oracle, .NET Compact Framework, "Качествен програмен код" и други. Цвятко участва в авторския екип на книгата "Въведение в програмирането в Java". Професионалните му интереси включват уеб и десктоп базирани технологии, клиентско ориентирани уеб технологии, бази данни и шаблони за дизайн.

Повече информация за него може да намерите на неговия блог: http://tsvyatkokonov.blogspot.com.

Редакторите

Освен авторите сериозен принос за създаването на книгата имат и редакторите, които участваха безвъзмездно в проверката на текста и примерите и отстраняването на грешки и други проблеми. Следват техните имена по азбучен ред:

- Веселин Георгиев
- Веселин Колев
- Дилян Димитров
- Дончо Минков
- Илиян Мурданлиев
- Йосиф Йосифов
- Марин Георгиев
- Мира Бивас
- Михаил Вълков
- Михаил Стойнов
- Николай Костов
- Николай Василев
- Павел Дончев
- Радослав Иванов

- Радослав Кирилов
- Радослав Тодоров
- Светлин Наков
- Станислав Златинов
- Стефан Стаев
- Теодор Божиков
- Цвятко Конов

Авторският колектив благодари и на Кристина Николова за нейните усилия по изработването на дизайна на корицата на книгата.

Книгата е безплатна!

Настоящата книга се разпространява напълно безплатно в електронен вид по лиценз, който позволява използването ѝ за всякакви цели, включително и в комерсиални проекти. Книгата се разпространява и в хартиен вид срещу заплащане, което покрива разходите по отпечатването и разпространението ѝ, без да се реализира печалба.

Отзиви

Ако не вярвате напълно на авторския колектив, разработил настоящата книга, може да се вдъхновите от отзивите за нея, дадени от водещи световни специалисти, включително софтуерни инженери от Майкрософт.

Отзив от Никола Михайлов, Microsoft

Програмирането е яко нещо! От стотици години хората се опитват да си направят живота по-лесен, за да работят по-малко. Програмирането позволява да се продължи тази тенденция към мързел на човечеството. Ако сте маниак на тема компютри, или просто искате да впечатлите останалите с един хубав сайт или нещо ваше "невиждано досега", добре дошли. Независимо дали сте от сравнително малката група "маниаци", които като видят хубава програма им се завърта главата, или просто искате да се реализирате професионално и да си живеете живота извън работа, тази книга е за вас.

Основните принципи на работа на двигател за коли не са се променили с години – гори там нещо (бензин, нафта или каквото сте сипали) и колата върви. По същия начин основните принципи на програмирането не са се променили от години насам. Дали ще пишете следващата игра, софтуер за управление на пари в банка или програмирате "мозъка" на новия биоробот, със сигурност ще използвате принципите и структурите от данни, описани в тази книга.

В книгата ще намерите голяма част от основите на програмирането. Аналогична фундаментална книга в автомобилната индустрия би била озаглавена "Двигатели с вътрешно горене".

Каквото и да правите, важното е да ви е приятно! Преди да започнете да четете тази книга – намислете си нещо за програмисти, което бихте искали да направите – било сайт, игра, или друга програма, която ви харесва! Докато прочитате книгата, мислете кое и как от прочетеното ще използвате за вашата програма! Ако ви е интересно, ще научите и най-сложното нещо с лекота!

Моята първа програма (с която се гордея достатъчно, за да говоря публично) беше просто рисуване по екрана със стрелките на клавиатурата. Доста време ми отне тогава да я направя, но като се получи ми хареса. Пожелавам ви и на вас: да ви харесва всичко свързано с програмирането! Приятно четене на книгата и успешна професионална реализация!

***Никола Михайлов** е софтуерен инженер в Майкрософт, в екипа разработващ Visual Studio. Автор на сайта http://nokola.com, лесно се "пали" на тема програмиране; винаги готов когато трябва да се пише нещо добро! Обича да помага на хора с въпроси и желание за програмиране, независимо дали са начинаещи или експерти. При нужда го потърсете по e-mail: nokola@nokola.com.*

Отзив от Васил Бакалов, Microsoft

"Въведение в програмирането със C#" е един смел опит не само да помогне на читателя да направи първите си стъпки в програмирането, а също да го запознае с програмната среда и тренира в практическите задачи, които възникват в ежедневието на програмиста. Авторите са намерили добро съчетание от теория, с която да предадат необходимите знания за писане и четене на програмен код, и практика – разнообразни задачи, подбрани да затвърдят знанията и да формират в читателя навика, че винаги, когато пишем програми, мислим не само за синтаксиса, който ще използваме, а и за ефективното решение на проблема.

Езикът C# е подходящ избор, защото е един елегантен език, с който не се тревожим за представянето на нашата програма в паметта на компютъра, а можем да се концентрираме да подобряваме ефективността и елегантността на нашата програма.

Досега не съм попадал на книга за програмиране, която едновременно да запознава читателя с езика и да формира уменията му за решаване на задачи. Радвам се, че сега има такава книга, и съм сигурен че ще бъде изключително полезна на бъдещите програмисти.

***Васил Бакалов** е софтуерен инженер в Microsoft Corporation, Redmond, участник в проекта за първата българска книга за .NET: "Програмиране за .NET Framework".* Неговият блог е достъпен от http://www.vassil.info.

Отзив от Васил Терзиев, Telerik

Преглеждайки книгата се върнах назад във времето, когато правех първи стъпки в PHP програмирането. Още си спомням книгата, от която се учех –

четирима автори, много неподредено и несвързано съдържание, елементарни примери в главите за напреднали и сложни примери в главите за начинаещи, различни конвенции, фокус изцяло върху платформата и езика, а не върху това как ефективно да ги използваме за писането на по-качествени приложения.

Много се радвам, че "Въведение в програмирането със C#" има съвсем различен подход. Нещата са обяснени много достъпно, но и с необходимата дълбочина и всяка една глава продължава и надгражда плавно предходните. Аз като страничен наблюдател съм бил свидетел на това какви усилия са положени за написването на книгата и съм щастлив, че този огромен заряд и желание да се създаде една по-различна книга, наистина се е материализирало в много качествено съдържание.

Силно се надявам, че тази книга ще бъде полезна за читателите и че ще им даде една добра основа, на която да стъпят, основа, която да ги запали към професионално развитие в областта на програмирането и която ще им помогне да направят един по-безболезнен и качествен старт.

***Васил Терзиев** е един от основателите и изпълнителен директор на Телерик АД, водещ производител на развойни средства и компоненти за .NET платформата на Майкрософт. Неговият блог е достъпен от адрес http://blogs.telerik.com/vassilterziev/. При желание, винаги може да се свържете с него по e-mail: terziev@telerik.com.*

Отзив от Веселин Райчев, Google

Може би и без да прочетете тази книга ще можете да работите като софтуерен разработчик, но смятам, че ще ви е много по-трудно.

Наблюдавал съм случаи на преоткриване на колелото, много често в по-лош вид от теоретично най-доброто и най-често целият екип губи от това. Всеки, занимаващ се с програмиране, рано или късно трябва да прочете какво е сложност на алгоритъм, какво е хеш-таблица, какво е двоично търсене или най-добрите практики за използване на шаблони за проектиране (design patterns). Защо не започнете още отсега като прочетете тази книга?

Съществуват много книги за C# и още повече за програмиране. За много от тях ще кажат, че са най-доброто ръководство, най-бързо навлизане в езика. Тази книга е различна с това, че ще ви покаже какво трябва да знаете, за да постигате успехи, а не какви са тънкостите на даден език за програмиране. Ако смятате темите в тази книга за безинтересни, вероятно софтуерното инженерство просто не е за вас.

***Веселин Райчев** е софтуерен инженер в Google, където се занимава с Google Maps и Google Translate. Преди това е работил в Motorola Biometrics и Metalife AG.*

Веселин е печелил призови отличия в редица национални и международни състезания и е носител на бронзов медал от Международната олимпиада по информатика, Южна Корея, 2002 и сребърен медал от Балкани-

ада по информатика. Два пъти е представял СУ "Св. Климент Охридски" на световни финали по информатика (ACM ICPC) и е преподавал в няколко изборни курса във Факултета по математика и информатика на СУ.

Отзив от Васил Поповски, VMware

Като служител с ръководна роля във фирма VMware и преди това в Sciant често ми се налага да правя технически интервюта на кандидати за работа в нашата фирма. Учудващо е колко голяма част от кандидатите за софтуерни инженери, които идват на интервюта при нас, не владеят фундаментални основи на програмирането. Случва се кандидати с дългогодишен опит да не могат да нарисуват свързан списък, да не знаят как работи хеш-таблицата, да не са чували какво е сложност на алгоритъм, да не могат да сортират масив или да го сортират, но със сложност O(n^3). Направо не е за вярване колко много самоуки програмисти има, които не владеят фундаменталните основи на програмирането, които ще намерите в тази книга. Много от практикуващите професията софтуерен разработчик не са наясно дори с най-основните структури от данни в програмирането и не знаят как да обходят дърво с рекурсия. За да не бъдете като тях, прочетете тази книга! Тя е първото учебно пособие, от което трябва да започнете своето развитие като програмисти. Фундаменталните познания по структури от данни, алгоритми и решаване на задачи, които ще намерите в тази книга, ще са ви необходими, за да изградите успешно кариерата си на софтуерен разработчик и разбира се, да бъдете успешни по интервютата за работа и след това на работното си място.

Ако започнете от правене на динамични уеб сайтове с бази от данни и AJAX, без да знаете какво е свързан списък, дърво или хеш-таблица, един ден ще разберете какви фундаментални пропуски в знанията си имате. Трябва ли да се изложите на интервю за работа, пред колегите си или пред началника си, когато се разбере, че не знаете за какво служи хеш-кодът или как работи структурата **`List<T>`** или как се обхождат рекурсивно директориите по твърдия диск?

Повечето книги за програмиране ще ви научат да пишете прости програмки, но няма да обърнат внимание на качеството на програмния код. Това е една тема, която повечето автори смятат за маловажна, но писането на качествен код е основно умение, което отличава кадърните от посредствените програмисти. С годините можете и сами да стигнете до добрите практики, които тази книга ще ви препоръча, но трябва ли да се учите по метода на пробите и грешките? Тази книга ще ви даде лесния начин да тръгнете в правилната посока – да овладеете базовите структури от данни и алгоритми, да се научите да мислите правилно и да пишете кода си качествено. Пожелавам ви ползотворно четене.

***Васил Поповски** е софтуерен архитект във VMware България с повече от 10 години професионален опит като Java разработчик. Във VMware България се занимава с разработка на скалируеми, Enterprise Java системи. Преди това е работил като старши мениджър във VMware*

България, като технически директор във фирма Sciant и като ръководител екип в SAP Labs България.

Като ученик Васил е печелил призови отличия в редица национални и международни състезания и е носител на бронзов медал от Международната олимпиада по информатика, Сетубал, 1998 и бронзов медал от Балканиада по информатика, Драма, 1997. Като студент Васил участвал в редица национални студентски състезания и в световното междууниверситетско състезание по програмиране (ACM ICPC). През 2001/2002 води курса "Обработване на транзакции" в СУ "Св. Климент Охридски". Васил е един от учредителите на Българска асоциация на разработчиците на софтуер (БАРС).

Отзив от Павлин Добрев, ProSyst Labs

Книгата "Въведение в програмирането със C#" е отлично учебно пособие за начинаещи, което ви дава възможност по лесен и достъпен начин да овладеете основите на програмирането. Това е шестата книга, написана под ръководството на Светлин Наков, и също както останалите, е изключително ориентирана към усвояването на практически умения за програмиране. Учебното съдържание обхваща фундаментални теми като структури от данни, алгоритми и решаване на задачи и това я прави непреходна при развитието на технологиите. Тя е изпълнена с многобройни примери и практически съвети за решаване на основни задачи от ежедневната работа на един програмист.

Книгата "Въведение в програмирането със C#" представлява адаптация към езика C# и платформата Microsoft .NET Framework на изключително успешната книга "Въведение в програмирането с Java" и се базира на натрупания опит на водещия автор Светлин Наков в преподаването на основи на програмирането – както в Национална академия по разработка на софтуер (НАРС) и по-късно в Telerik Academy, така и във ФМИ на Софийски университет "Св. Климент Охридски", Нов български университет (НБУ) и Технически университет-София.

Въпреки големия брой автори, всеки от които с различен професионален и преподавателски опит, между отделните глави на книгата се забелязва ясна логическа свързаност. Тя е написана разбираемо, с подробни обяснения и с много, много примери, далеч от сухия академичен стил, присъщ за повечето университетски учебници.

Насочена към прохождащите в програмирането, книгата поднася внимателно, стъпка по стъпка, най-важното, което един програмист трябва да владее, за да практикува професията си – започвайки от променливи, цикли и масиви и достигайки до фундаменталните структури от данни и алгоритми. Книгата засяга и важни теми като рекурсивни алгоритми, дървета, графи и хеш-таблици. Това е една от малкото книги, която същевременно учи на добър програмен стил и качествен програмен код. Отделено е достатъчно внимание на принципите на обектно-ориентираното програмиране и обработката на изключения, без които съвременната разработка на софтуер е немислима.

Книгата "Въведение в програмирането със C#" учи на важните принципи и концепции в програмирането, на начина, по който програмистите разсъждават логически, за да решават проблемите, с които се сблъскват в ежедневната си работа. Ако трябваше заглавието на книгата да съответства още по-точно на съдържанието й, тя трябваше да се казва "Фундаментални основи на програмирането".

Тази книга не съдържа всичко за програмирането и няма да ви направи .NET софтуерни инженери. За да станете наистина добри програмисти, ви трябва много, много практика. Започнете от задачите за упражнения след всяка глава, но не се ограничавайте само с тях. Ще изпишете хиляди редове програмен код докато наистина станете добри – такъв е животът на програмиста. Тази книга е наистина силен старт! Възползвайте се от възможността да намерите всичко най-важно на куп, без да се лутате из хилядите самоучители и статии в Интернет. На добър път!

Д-р Павлин Добрев *е технически директор на фирма Просист Лабс (www.prosyst.com), софтуерен инженер с повече от 15 години опит, консултант и учен, доктор по Компютърни системи, комплекси и мрежи. Павлин има световен принос в развитието на съвременните компютърни технологии и технологични стандарти. Той участва активно в международни стандартизационни организации като OSGi Alliance (www.osgi.org) и Java Community Process (www.jcp.org), както и инициативи за софтуер с отворен код като Eclipse Foundation (www.eclipse.org). Павлин управлява софтуерни проекти и консултира фирми като Miele, Philips, Siemens, BMW, Bosch, Cisco Systems, France Telecom, Renault, Telefonica, Telekom Austria, Toshiba, HP, Motorola, Ford, SAP и др. в областта на вградени приложения, OSGi базирани системи за автомобили, мобилни устройства и домашни мрежи, среди за разработка и Java Enterprise сървъри за приложения. Той има много научни и технически публикации и е участник в престижни международни конференции.*

Отзив от Николай Манчев, Oracle

За да станете добър разработчик на софтуер, трябва да имате готовност да инвестирате в натрупването на познания в няколко области и конкретния език за програмиране е само една от тях. Добрият разработчик трябва да познава не само синтаксиса и приложно-програмния интерфейс на езика, който си е избрал. Той трябва да притежава също така задълбочени познания по обектно-ориентирано програмиране, структури от данни и писане на качествен код. Той трябва да подкрепи тези си познания и със сериозен практически опит.

Когато започвах своята кариера на разработчик на софтуер преди повече от 15 години, намирането на цялостен източник, от който да науча тези неща беше невъзможно. Да, тогава имаше книги за отделните програмни езици, но те описваха единствено техния синтаксис. За описание на приложно-програмния интерфейс трябваше да се ползва самата документация към библиотеките. Имаше отделни книги посветени единствено на обектно-ориентираното програмиране. Различни алгоритми и структури от данни пък се преподаваха в университета. За качествен програмен код не се говореше въобще.

Научаването на всички тези неща "на парче" и усилията по събирането им в единен контекст си оставаше работа на избралия "пътя на програмиста". Понякога един такъв самообразоващ се програмист не успява да запълни огромни пропуски в познанията си просто защото няма идея за тяхното съществуване. Нека ви дам един пример, за да илюстрирам проблема.

През 2000 г. поех да управлявам един голям Java проект. Екипът, който го разработваше беше от 25 души и до момента по проекта имаше написани приблизително 4 000 Java класа. Като ръководител на екипа, част от моята работа включваше редовното преглеждане на кода написан от другите програмисти. Един ден видях как един от моите колеги беше решил стандартната задача по сортиране на масив. Той беше написал отделен метод от около 25 реда, който реализираше тривиалния алгоритъм за сортиране по метода на мехурчето. Когато отидох при него и го запитах защо е направил това вместо да реши проблема на един единствен ред използвайки **`Arrays.sort()`**, той се впусна в обяснения как вградения метод е по-тромав и е по-добре тези неща да си ги пишеш сам. Накарах го да отвори документацията и му показах, че "тромавият" метод работи със сложност O(n*log(n)), а неговото мехурче е еталон за лоша производителност със своята сложност O(n*n). В следващите няколко минути от нашия разговор направих и истинското откритие – моят колега нямаше идея какво е сложност на алгоритъма, а самите му познания по стандартни алгоритми бяха трагични. В последствие открих, че той е завършил съвсем друг тип инженерна специалност, а не информатика. В това, разбира се, няма абсолютно нищо лошо. В познанията си по Java той не отстъпваше на останалите колеги, които имаха по-дълъг практически опит от него. Но в този ден ние открихме празнина в неговата квалификация на разработчик, за която той не беше и подозирал.

Не искам да оставате с погрешни впечатления от тази история. Въпреки, че един студент издържал успешно основните си изпити по специалност "Информатика" в добър университет със сигурност ще знае базовите алгоритми за сортиране и ще може да изчисли тяхната сложност, той също ще има своите пропуски. Тъжната истина е, че в България университетското образование по тази специалност все още е с твърде теоретична насоченост. То твърде малко се е променило за последните 15 години. Да, програмите вече се пишат на Java и C#, но това са същите програми, които се пишеха тогава на Pascal и Ada.

Преди около година приех за консултация студент първокурсник, който следваше в специалност "Информатика" на един от най-големите държавни университети в България. Когато седнахме да прегледаме заедно записките му от лекциите по "Увод в програмирането" бях изумен от примерния код даван от преподавателя. Имената на методите бяха смесица от английски и транслитериран български. Имаше метод `calculate` и метод `rezultat`. Променливите носеха описателните имена `a1`, `a2`, и `suma`. Да, в този подход няма нищо трагично, докато се използва за примери от десет реда, но когато този студент заеме след години своето заслужено място в някой голям проект, той ще бъде тежко порицан от ръководителя на проекта, който ще му обяснява за код конвенции, именуване на променливи, методи и класове, логическа свързаност на отговорностите и диапазон на активност. Тогава те заедно ще открият неговата празнина в познанията по качествен код по същия начин, по който ние с моя колега открихме проблемните му познания в областта на алгоритмите.

Скъпи читателю, смело мога да заявя, че в ръцете си държиш една наистина уникална книга. Нейното съдържание е подбрано изключително внимателно. То е подредено и поднесено с внимание към детайлите, на който са способни само хора с огромен практически опит и солидни научни познания като водещите автори на тази книга Светлин Наков и Веселин Колев. Години наред те също са се учили "в движение", допълвайки и разширявайки своите познания. Работили са години по огромни софтуерни проекти, участвали са в научни конференции, преподавали са на стотици студенти. Те знаят какво е нужно да знае всеки един, който се стреми към кариера в областта на разработката на софтуер и са го поднесли така, както никоя книга по увод в програмирането не го е правила до момента. Твоето пътуване през страниците ще те преведе през синтаксиса на езика C#. Ще видиш използването на голяма част от приложно-програмния му интерфейс. Ще научиш основите на обектно-ориентираното програмиране и ще боравиш свободно с термини като обекти, събития и изключения. Ще видиш най-често използваните структури от данни като масиви, дървета, хеш-таблици и графи. Ще се запознаеш с най-често използваните алгоритми за работа с тези структури и ще узнаеш за техните плюсове и минуси. Ще разбереш концепциите по конструиране на качествен програмен код и ще знаеш какво да изискваш от програмистите си, когато някой ден станеш ръководител на екип. В допълнение книгата ще те предизвика с много практически задачи, които ще ти помогнат да усвоиш по-добре и по пътя на практиката материала, който се разглежда в нея. А ако някоя от задачите те затрудни, винаги ще можеш да погледнеш решението, което авторите предоставят за всяка от тях.

Програмистите правят грешки – от това никой не е застрахован. По-добрите грешат от недоглеждане или преумора, а по-лошите – от незнание. Дали ще станеш добър или лош разработчик на софтуер зависи изцяло от теб и най-вече от това, доколко си готов постоянно да инвестираш в своите познания – било чрез курсове, чрез четене или чрез практическа работа. Със сигурност обаче мога да ти кажа едно – колкото и време да инвестираш в тази книга, няма да сгрешиш. Ако преди няколко години някой, желаещ да стане разработчик на софтуер, ме попиташе "От къде да започна?" нямаше как да му дам еднозначен отговор. Днес мога без притеснения да заявя - "Започни от тази книга (във варианта й за C# или Java)!".

От все сърце ти желая успех в овладяването на тайните C#, .NET Framework и разработката на софтуер!

Николай Манчев *е консултант и софтуерен разработчик с дългогодишен опит в Java Enterprise и Service Oriented Architecture (SOA). Работил е за BEA Systems и Oracle Corporation. Той е сертифициран разработчик по програмите на Sun, BEA и Oracle. Преподава софтуерни технологии и води курсове по Мрежово програмиране, J2EE, Компресия на данни и Качествен програмен код в ПУ "Паисий Хилендарски" и СУ "Св. Климент Охридски". Водил е редица курсове за разработчици по Oracle технологии в централна и източна Европа (Унгария, Гърция, Словакия, Словения, Хърватска и други) и е участвал в международни проекти по внедряване на J2EE базирани системи за управление на сигурността. Негови разработки в областта на алгоритмите за компресия на данни са приети и представяни в САЩ от IEEE. Николай е почетен член на Българска асоциация на разработчиците на софтуер (БАРС). Автор е на книгата "Сигурност в Oracle Database : Версия 10g и 11g". Повече за него можете да намерите на личния му уеб сайт: http://www.manchev.org. За да се свържете с него използвайте e-mail: nick@manchev.org.*

Отзив от Панайот Добриков, SAP AG

Настоящата книга е едно изключително добро въведение в програмирането за начинаещи и водещ пример в течението (промоцирано от Wikipedia и други) да се създава и разпространява достъпно за всеки знание не само *безплатно*, но и с изключително високо качество.

Панайот Добриков *е програмен директор в SAP AG и автор на книгата "Програмиране=++Алгоритми;". Повече за него можете да намерите на личния му уеб сайт: http://indyana.hit.bg.*

Отзив от Любомир Иванов, Mobiltel

Ако преди 5 или 10 години някой ми беше казал, че съществува книга, от която да научим основите на управлението на хора и проекти – бюджетиране, финанси, психология, планиране и т.н., нямаше да му повярвам. Не бих повярвал и днес. За всяка от тези теми има десетки книги, които трябва да бъдат прочетени.

Ако преди година някой ми беше казал, че съществува книга, от която можем да научим основите на програмирането, необходими на всеки софтуерен разработчик – пак нямаше да му повярвам.

Спомням си времето като начинаещ програмист и студент – четях няколко книги за езици за програмиране, други за алгоритми и структури от данни, а трети за писане на качествен код. Много малко от тях ми помогнаха да мисля алгоритмично и да си изградя подход за решаване на ежедневните проблеми, с които се сблъсквах в практиката. Нито една не ми даде цялостен поглед над всичко, което исках и трябваше да знам като програмист и софтуерен инженер. Единственото, което помагаше, беше инатът и преоткриването на колелото.

Днес чета тази книга и се радвам, че най-сетне, макар и малко късно за мен, някой се е хванал и е написал Книгата, която ще помогне на всеки начинаещ програмист да сглоби големия пъзел на програмирането – модерен език за програмиране, структури от данни, качествен код, алгоритмично мислене и решаване на проблеми. Това е книгата, от която трябва да за почнете с програмирането, ако искате да овладеете изкуството на качественото програмиране. Дали ще изберете C# или Java варианта на тази книга няма особено значение. Важното е да се научите да мислите като програмисти и да решавате проблемите, които възникват при писането на софтуер, а езикът е само един инструмент, който можете да смените с друг по всяко време.

Тази книга не е само за начинаещите. Дори програмисти с няколкогодишен опит има какво да научат от нея. Препоръчвам я на всеки разработчик на софтуер, който би искал да разбере какво не е знаел досега.

Приятно четене!

Любомир Иванов *е ръководител на отдел "Data and Mobile Applications" в Мобилтел ЕАД, където се занимава с разработка и внедряване на ИТ решения за telecom индустрията.*

Отзив от Христо Дешев

Учудващо е, че голям процент от програмистите не обръщат внимание на малките неща като имената на променливите и добрата структура на кода. Тези неща се натрупват и накрая формират разликата между добре написания софтуер и купчината спагети. Тази книга учи на дисциплина и "хигиена" в писането на код още с основите на програмирането, а това несъмнено ще Ви изгради като професионалист.

Христо Дешев*, software craftsman*

Спонсор

Авторският колектив благодари на спонсора на книгата – иновативната софтуерна компания Telerik (www.telerik.com) – която подпомогна издаването на книгата на хартия и отдели от работното време на свои служители, които да участват безвъзмездно за реализирането на проекта.

telerik

deliver more than expected

Telerik Corporation е водещ производител на ASP.NET AJAX, Silverlight, Windows Forms и WPF компоненти и решения за създаване на отчети (reporting), обектно-релационни технологии (ORM), системи за управление на уеб съдържание (CMS) за .NET платформата, гъвкави инструменти за управление на проекти (agile project management), инструменти за автоматизирано тестване, добавки към Visual Studio за подобряване на удобството при разработка и множество други продукти и технологии. Телерик е българска продуктово-ориентирана иновативна технологична компания със седалище в София и офиси в САЩ, Канада, Великобритания, Германия и Австралия, златен партньор на Microsoft. В компанията работят повече от 400 служители, повечето от които софтуерни инженери. Заради отлична работна среда и високи постижения Телерик става работодател номер едно на България за 2007 г. и 2010 г. (глобално, за всички индустрии) и е един от най-добрите работодатели в централна източна Европа.

Благодарение на Telerik Corporation настоящата книга ще бъде достъпна в хартиен вид на цена, покриваща разходите по нейното отпечатване и разпространение, без да се начислява печалба. Надяваме се това да позволи и на колеги с по-ниски материални възможности да я прибавят към личната си библиотека.

Ако все пак тази книга наистина ви трябва и сте изключително силно мотивирани да я прочетете и да решите всички задачи от нея, но нямате финансова възможност да си закупите нейния хартиен вариант, изпратете своята история във вид на мотивационно писмо до academy@telerik.com и може да ви доставим безплатно хартиено копие.

Лиценз

Книгата и учебните материали към нея се разпространяват свободно по следния лиценз:

Общи дефиниции

1. Настоящият лиценз дефинира условията за използване и разпространение на учебни материали и книга "Въведение в програмирането със C#", разработени от екип под ръководството на Светлин Наков (www.nakov.com) и Веселин Колев (veskokolev.blogspot.com).

2. Учебните материали се състоят от:

   - книга (учебник) по "Въведение в програмирането със C#";
   - примерен сорс-код;
   - демонстрационни програми;
   - задачи за упражнения;

3. Учебните материали са достъпни за свободно изтегляне при условията на настоящия лиценз от официалния сайт на проекта:

http://www.introprogramming.info

4. Автори на учебните материали са лицата, взели участие в тяхното изработване.
5. Потребител на учебните материали е всеки, който по някакъв начин използва тези материали или части от тях.

Права и ограничения на потребителите

3. Потребителите **имат** право:
   - да разпространяват безплатно непроменени копия на учебните материали в електронен или хартиен вид;
   - да използват учебните материали или части от тях, включително примерите и демонстрациите, включени към учебните материали или техни модификации, за всякакви нужди, включително и в комерсиални проекти, при условие че ясно посочват оригиналния източник, оригиналния автор на съответния текст или програмен код, настоящия лиценз и сайта www.introprogramming.info;
   - да разпространяват безплатно извадки от учебните материали или техни модифицирани копия (включително да ги превеждат на чужди езици или да ги адаптират към други програмни езици и платформи), но само при изричното споменаване на оригиналния първоизточник и авторите на съответния текст, програмен код или друг материал, настоящия лиценз и официалния сайт на проекта – www.introprogramming.info.
4. Потребителите **нямат** право:
   - да разпространяват срещу заплащане учебните материали или части от тях, като изключение прави само програмният код;
   - да премахват настоящия лиценз от учебните материали, когато ги модифицират за свои нужди.

Права и ограничения на авторите

1. Всеки автор притежава неизключителни права върху продуктите на своя труд, с които взима участие в изработката на учебните материали.
2. Авторите имат право да използват частите, изработени от тях, за всякакви цели, включително да ги изменят и разпространяват срещу заплащане.
3. Правата върху учебните материали, изработени в съавторство, са притежание на всички съавтори заедно.
4. Авторите нямат право да разпространяват срещу заплащане учебни материали или части от тях, изработени в съавторство, без изричното съгласие на всички съавтори.

Сайтът на книгата

Официалният уеб сайт на книгата "Въведение в програмирането със C#" е достъпен от адрес: http://www.introprogramming.info. От него можете да изтеглите цялата книга в електронен вид, сорс кода на примерите и други полезни ресурси.

Дискусионна група

Дискусионната група, в която можете да намерите решение на почти всички задачи от книгата е достъпна в Google Groups от следния адрес: http://groups.google.com/group/telerikacademy. Тази група е създадена за дискусия между участниците в курсовете от Telerik Academy, които в първите няколко месеца на своето обучение преминават през целия учебен материал от настоящата книга и задължително решават всички задачи от упражненията.

В групата ще намерите както коментари и решения, изпратени както от студенти и читатели на книгата, така и от авторитетни преподаватели от Академията. Просто се разровете достатъчно задълбочено в архивите на групата и ще намерите по няколко решения на всички задачи от книгата (без изключения). Всяка година няколко стотици участници в курсовете на Телерик решават всички задачи от тази книга и споделят решенията и трудностите, с които са се сблъскали в групата, така че просто търсете усърдно в архивите, ако не можете да се справите с някоя задача.

Видеоматериали за самообучение по книгата

В рамките на програмата Telerik Academy (http://academy.telerik.com) е направен видеозапис на всички лекции, разработени по учебното съдържание на настоящата книга в рамките на безплатния курс "Fundamentals of C# Programming". Видеоматериалите са достъпни от сайта на Telerik Academy – http://academy.telerik.com (разгледайте безплатния курс "C# Fundamentals").

Фен клуб

Фен клубът на книгата "Въведение в програмирането със C#" е организиран като група в социалната мрежа за бизнес контакти LinkedIn: http://www.linkedin.com/groupInvitation?groupID=1724867.

Светлин Наков,
Ръководител на отдел "технологично обучение",
Telerik Corporation,
26.06.2011 г.

Глава 1. Въведение в програмирането

В тази тема...

В настоящата тема ще разгледаме основните термини от програмирането и ще напишем първата си програма на C#. Ще се запознаем с това какво е програмиране и каква е връзката му с компютрите и програмните езици.

Накратко ще разгледаме основните етапи при писането на софтуер.

Ще въведем езика C# и ще се запознаем с .NET платформата и технологиите на Microsoft за разработка на софтуер. Ще разгледаме какви помощни средства са ни необходими, за да можем да програмираме на C#. Ще използваме езика C#, за да напишем първата си програма, ще я компилираме и изпълним както от командния ред, така и от средата за разработка Microsoft Visual Studio 2010 Express Edition. Ще се запознаем още и с MSDN Library – документацията на .NET Framework, която ни помага при по-нататъшно изследване на възможностите на езика и платформата.

Какво означава "да програмираме"?

В днешно време компютрите навлизат все по-широко в ежедневието ни и все повече имаме нужда от тях, за да се справяме със сложните задачи на работното място, да се ориентираме, докато пътуваме, да се забавляваме или да общуваме. Неизброимо е приложението им в бизнеса, в развлекателната индустрия, в далекосъобщенията и в областта на финансите. Няма да преувеличим, ако кажем, че компютрите изграждат нервната система на съвременното общество и е трудно да си представим съществуването му без тях.

Въпреки масовото им използване, малко хора имат представа как всъщност работят компютрите. На практика не компютрите, а програмите, които се изпълняват върху тях (софтуерът), имат значение. Този софтуер придава стойността за потребителите и чрез него се реализират различните типове услуги, променящи живота ни.

Как компютрите обработват информация?

За да разберем какво значи да програмираме, нека грубо да сравним компютъра и операционната система, работеща на него, с едно голямо предприятие заедно с неговите цехове, складове и транспортни механизми. Това сравнение е грубо, но дава възможност да си представим степента на сложност на един съвременен компютър. В компютъра работят много процеси, които съответстват на цеховете и поточните линии в предприятието. Твърдият диск заедно с файловете на него и оперативната (RAM) памет съответстват на складовете, а различните протоколи са транспортните системи, внасящи и изнасящи информация.

Различните видове продукция в едно предприятие се произвеждат в различните цехове. Цеховете използват суровини, които взимат от складовете, и складират готовата продукция обратно в тях. Суровините се транспортират в складовете от доставчиците, а готовата продукция се транспортира от складовете към пласмента. За целта се използват различни видове транспорт. Материалите постъпват в предприятието, минават през различни стадии на обработка и напускат предприятието, преобразувани под формата на продукти. Всяко предприятие преобразува суровините в готов за употреба продукт.

Компютърът е машина за обработка на информация и при него както суровината, така и продукцията е информация. Входната информация най-често се взима от някой от складовете (файлове или RAM памет), където е била транспортирана, преминава през обработка от един или повече процеси и излиза модифицирана като нов продукт. Пример за това са уеб базираните приложенията. При тях за транспорт както на суровините, така и на продукцията, се използва протоколът HTTP, а обработката на информация обикновено е свързана с извличане на съдържание от база данни и подготовката му за визуализация във вид на HTML.

Управление на компютъра

Целият процес на изработка на продуктите в едно предприятие има много степени на управление. Отделните машини и поточни линии се управляват от оператори, цеховете се управляват от управители, а предприятието като цяло се управлява от директори. Всеки от тях упражнява контрол на различно ниво. Най-ниското ниво е това на машинните оператори – те управляват машините, образно казано, с помощта на копчета и ръчки. Следващото ниво е на управителите на цехове. На най-високо ниво са директорите, те управляват различните аспекти на производствените процеси в предприятието. Всеки от тях управлява, като издава заповеди.

По аналогия при компютрите и софтуера има много нива на управление. На най-ниско машинно ниво се управлява самият процесор и регистрите му (чрез машинни програми на ниско ниво) – можем да сравним това с управлението на машините в цеховете. На по-високо системно ниво се управляват различните отговорности на операционната система (например Windows 7) като файлова система, периферни устройства, потребители, комуникационни протоколи – можем да сравним това с управлението на цеховете и отделите в предприятието. На най-високо ниво в софтуера са приложенията (приложните програми). При тях се управлява цял ансамбъл от процеси, за изпълнението на които са необходими огромен брой операции на процесора. Това е нивото на директорите, които управляват цялото предприятие с цел максимално ефективно използване на ресурсите за получаване на качествени резултати.

Същност на програмирането

Същността на програмирането е да се управлява работата на компютъра на всичките му нива. Управлението става с помощта на "заповеди" и "команди" от програмиста към компютъра, известни още като програмни инструкции. Да програмираме, означава да организираме управлението на компютъра с помощта на поредици от инструкции. Тези заповеди (инструкции) се издават в писмен вид и биват безпрекословно изпълнявани от компютъра (съответно от операционната система, от процесора и от периферните устройства).

Програмистите са хората, които създават инструкциите, по които работят компютрите. Тези инструкции се наричат **програми**. Те са много на брой и за изработката им се използват различни видове програмни езици. Всеки език е ориентиран към някое ниво на управление на компютъра. Има езици, ориентирани към машинното ниво – например асемблер, други са ориентирани към системното ниво (за взаимодействие с операционната система), например C. Съществуват и езици от високо ниво, ориентирани към писането на приложни програми. Такива са езиците C#, Java, C++, PHP, Visual Basic, Python, Ruby, Perl и други.

В настоящата книга ще разгледаме програмния език C#, който е съвременен език за програмиране от високо ниво. При използването му позицията на програмиста в компютърното предприятие се явява тази на директора. Инструкциите, подадени като програми на C# могат да имат достъп и да управляват почти всички ресурси на компютъра директно или посредством операционната система. Преди да разгледаме как може да се използва C# за писане на прости компютърни програми, нека разгледаме малко по-широко какво означава да разработваме софтуер, тъй като програмирането е най-важната дейност в този процес, но съвсем не е единствената.

Етапи при разработката на софтуер

Писането на софтуер може да бъде сложна задача, която отнема много време на цял екип от софтуерни инженери и други специалисти. Затова с времето са се обособили различни методики и практики, които улесняват живота на програмистите. Общото между всички тях е, че разработката на всеки софтуерен продукт преминава през няколко етапа, а именно:

- Събиране на изискванията за продукта и изготвяне на задание;
- Планиране и изготвяне на архитектура и дизайн;
- Реализация (включва писането на програмен код);
- Изпитания на продукта (тестове);
- Внедряване и експлоатация;
- Поддръжка.

Фазите реализация, изпитания, внедряване и поддръжка се осъществяват в голямата си част с помощта на програмиране.

Събиране на изискванията и изготвяне на задание

В началото съществува само идеята за определен продукт. Тя включва набор от изисквания, дефиниращи действия от страна на потребителя и компютъра, които в общия случай улесняват извършването на досега съществуващи дейности. Като пример може да дадем изчисляването на заплатите, пресмятане на балистични криви, търсене на най-пряк път в Google Maps. Много често софтуерът реализира несъществуваща досега функционалност като например автоматизиране на някаква дейност.

Изискванията за продукта обикновено се дефинират под формата на документи, написани на естествен език – български, английски или друг. На този етап не се програмира. Изискванията се дефинират от експерти, запознати с проблематиката на конкретната област, които умеят да ги описват в разбираем за програмистите вид. В общия случай тези експерти не са специалисти по програмиране и се наричат **бизнес анализатори**.

Планиране и изготвяне на архитектура и дизайн

След като изискванията бъдат събрани, идва ред на етапа на планиране. През този етап се съставя технически план за изпълнението на проекта, който описва платформите, технологиите и първоначалната архитектура (дизайн) на програмата. Тази стъпка включва значителна творческа работа и обикновено се реализира от софтуерни инженери с много голям опит, наричани понякога **софтуерни архитекти**. Съобразно изискванията се избират:

- Вида на приложението – например конзолно приложение, настолно приложение (GUI, Graphical User Interface application), клиент-сървър приложение, уеб приложение, Rich Internet Application (RIA) или peer-to-peer приложение;
- Архитектурата на програмата – например еднослойна, двуслойна, трислойна, многослойна или SOA архитектура;
- Програмният език, най-подходящ за реализирането – например C#, Java или C++, или комбинация от езици;
- Технологиите, които ще се ползват: платформа (примерно Microsoft .NET, Java EE, LAMP или друга), сървър за бази данни (примерно Oracle, SQL Server, MySQL или друг), технологии за потребителски интерфейс (примерно Flash, JavaServer Faces, Eclipse RCP, ASP.NET, Windows Forms, Silverlight, WPF или други), технологии за достъп до данни (примерно Hibernate, JPA или LINQ-to-SQL), технологии за изготвяне на отчети (примерно SQL Server Reporting Services, Jasper Reports или други) и много други технологии и комбинации от технологии, които ще бъдат използвани за реализирането на различни части от софтуерната система.
- Броят и уменията на хората, които ще съставят екипа за разработка (големите и сериозни проекти се пишат от големи и сериозни екипи от разработчици);
- План на разработката – етапи, на които се разделя функционалността, ресурси и срокове за изпълнението на всеки етап.
- Други (големина на екипа, местоположение на екипа, начин на комуникация и т.н.).

Въпреки че съществуват много правила, спомагащи за правилния анализ и планиране, на този етап се изискват значителна интуиция и усет. Тази стъпка предопределя цялостното по-нататъшно развитие на процеса на разработка. На този етап не се извършва програмиране, а само подготовка за него.

Реализация

Етапът, най-тясно свързан с програмирането, е етапът на реализацията (имплементацията). На този етап съобразно заданието, дизайна и архитектурата на програмата (приложението) се пристъпва към реализирането (написването) й. Етапът "реализация" се изпълнява от **програмисти**, които пишат програмния код (сорс кода). При малки проекти останалите етапи могат да бъдат много кратки и дори да липсват, но етапът на реализация винаги се извършва, защото иначе не се изработва софтуер.

Настоящата книга е посветена главно на описание на средствата и похватите, използвани на този етап – изграждане на програмистско мислене и използване на средствата на езика C# и платформата .NET Framework за реализация на софтуерни приложения.

Изпитания на продукта (тестове)

Важен етап от разработката на софтуер е етапът на изпитания на продукта. Той цели да удостовери, че реализацията следва и покрива изискванията на заданието. Този процес може да се реализира ръчно, но предпочитаният вариант е написването на автоматизирани тестове, които да реализират проверките. Тестовете са малки програми, които автоматизират, до колкото е възможно, изпитанията. Съществуват парчета функционалност, за които е много трудно да се напишат тестове и поради това процесът на изпитание на продукта включва както автоматизирани, така и ръчни процедури за проверка на функционалността и качеството.

Процесът на тестване (изпитание) се реализира от екип **инженери по осигуряването на качеството** – quality assurance (QA) инженери. Те работят в тясно взаимодействие с програмистите за откриване и коригиране на дефектите (бъговете) в софтуера. На този етап почти не се пише нов програмен код, а само се отстраняват дефекти в съществуващия код.

В процеса на изпитанията най-често се откриват множество пропуски и грешки и програмата се връща обратно в етап на реализация. До голяма степен етапите на реализация и изпитания вървят ръка за ръка и е възможно да има множество преминавания между двете фази преди продуктът да е покрил изискванията на заданието и да е готов за етапа на внедряване и експлоатация.

Внедряване и експлоатация

Внедряването или инсталирането (deployment) е процесът на въвеждане на даден софтуерен продукт в експлоатация. Ако продуктът е сложен и обслужва много хора, този процес може да се окаже най-бавният и най-скъпият. За по-малки програми това е относително бърз и безболезнен процес. Най-често се разработва специална програма – инсталатор, която спомага за по-бързата и лесна инсталация на продукта. Понякога, ако продуктът се внедрява в големи корпорации с десетки хиляди копия, се разработва допълнителен поддържащ софтуер специално заради внедряването. След като внедряването приключи, продуктът е готов за експлоатация и следва обучение на служителите как да го ползват.

Като пример можем да дадем внедряването на Microsoft Windows в българската държавна администрация. То включва инсталиране и конфигуриране на софтуера и обучение на служителите.

Внедряването се извършва обикновено от екипа, който е разработил продукта или от специално обучени **специалисти по внедряването**. Те могат да бъдат системни администратори, администратори на бази данни

(DBA), системни инженери, специализирани консултанти и други. В този етап почти не се пише нов код, но съществуващият код може да се доработва и конфигурира докато покрие специфичните изисквания за успешно внедряване.

Поддръжка

В процеса на експлоатация неминуемо се появяват проблеми – заради грешки в самия софтуер или заради неправилното му използване и конфигурация или най-често заради промени в нуждите на потребителите. Тези проблеми довеждат до невъзможност за решаване на бизнес задачите чрез употреба на продукта и налагат допълнителна намеса от страна на разработчиците и експертите по поддръжката. Процесът по поддръжка обикновено продължава през целия период на експлоатация независимо колко добър е софтуерният продукт.

Поддръжката се извършва от екипа по разработката на софтуера и от специално обучени **експерти по поддръжката**. В зависимост от промените, които се правят, в този процес могат да участват бизнес анализатори, архитекти, програмисти, QA инженери, администратори и други.

Ако например имаме софтуер за изчисление на работни заплати, той ще има нужда от актуализация при всяка промяна на данъчното законодателство, което касае обслужвания счетоводен процес. Намеса на екипа по поддръжката ще е необходима и например ако бъде сменен хардуерът, използван от крайните клиенти, защото програмата ще трябва да бъде инсталирана и конфигурирана наново.

Документация

Етапът на документацията всъщност не е отделен етап, а съпътства всички останали етапи. Документацията е много важна част от разработката на софтуер и цели предаване на знания между различните участници в разработката и поддръжката на продукта. Информацията се предава както между отделните етапи, така и в рамките на един етап. Документацията обикновено се прави от самите разработчици (архитекти, програмисти, QA инженери и други) и представлява съвкупност от документи.

Разработката на софтуер не е само програмиране

Както сами се убедихте, разработването на софтуер не е само програмиране и включва много други процеси като анализ на изискванията, проектиране, планиране, тестване и поддръжка, в които участват не само програмисти, но и много други специалисти, наричани **софтуерни инженери**. Програмирането е само една малка, макар и много съществена, част от разработката на софтуера.

В настоящата книга ще се фокусираме само и единствено върху програмирането, което е единственото действие от изброените по-горе, без което не можем да разработваме софтуер.

Нашата първа C# програма

Преди да преминем към подробно описание на езика C# и на .NET платформата, нека да се запознаем с прост пример на това какво представлява една програма, написана на C#:

```
class HelloCSharp
{
   static void Main(string[] args)
   {
      System.Console.WriteLine("Hello C#!");
   }
}
```

Единственото нещо, което прави тази програма, е да изпише съобщението "Hello, C#!" на стандартния изход. Засега е още рано да я изпълняваме и затова само ще разгледаме структурата й. Малко по-нататък ще дадем пълно описание на това как да се компилира и изпълни дадена програма както от командния ред, така и от среда за разработка.

Как работи нашата първа C# програма?

Нашата първа програма е съставена от три логически части:

- Дефиниция на клас **HelloCSharp**;
- Дефиниция на метод **Main()**;
- Съдържание на метода **Main()**.

Дефиниция на клас

На първия ред от нашата програма дефинираме клас с името **HelloCSharp**. Най-простата дефиниция на клас се състои от ключовата дума **class**, следвана от името на класа. В нашия случай името на класа е **HelloCSharp**. Съдържанието на класа е разположено в блок от програмни редове, ограден във фигурални скоби: **{}**.

Дефиниция на метод Main()

На третия ред дефинираме метод с името **Main()**, която представлява входна или стартова точка за програмата. Всяка програма на C# се стартира от метод **Main()** със следната заглавна част (сигнатура):

```
static void Main(string[] args)
```

Методът трябва да е деклариран точно по начина, указан по-горе, трябва да е **static** и **void**, трябва да има име **Main** и като списък от параметри трябва да има един единствен параметър от тип масив от **string**. В нашия пример параметърът се казва **args**, но това не е задължително. Тъй като този параметър обикновено не се използва, той може да се пропусне. В такъв случай входната точка на програмата може да се опрости и да добие следния вид:

```
static void Main()
```

Ако някое от гореспоменатите изисквания не е спазено, програмата ще се компилира, но няма да може да се стартира, защото не е дефинирана коректно нейната входна точка.

Съдържание на Main() метода

Съдържанието на всеки метод се намира след сигнатурата на метода, заградено от отваряща и затваряща къдрави скоби. На следващия ред от примерната програма използваме системния обект **`System.Console`** и неговия метод **`WriteLine()`**, за да изпишем някакво съобщение в стандартния изход (на конзолата) в случая текста "Hello, C#!".

В **`Main()`** метода можем да напишем произволна последователност от изрази и те ще бъдат изпълнени в реда, в който сме ги задали.

Подробна информация за изразите може да се намери в главата "Оператори и изрази", работата с конзолата е описана в главата "Вход и изход от конзолата", а класовете и методите са описани подробно в главата "Дефиниране на класове".

C# различава главни от малки букви!

В горния пример използвахме някои ключови думи, като **`class`**, **`static`** и **`void`** и имената на някои от системните класове и обекти, като **`System.Console`**.

> **Внимавайте, докато пишете! Изписването на един и същ текст с главни, малки букви или смесено в C# означава различни неща. Да напишем** `Class` **е различно от** `class` **и да напишем** `System.Console` **е различно от** `SYSTEM.CONSOLE`**.**

Това правило важи за всички конструкции в кода – ключови думи, имена на променливи, имена на класове и т.н.

Програмният код трябва да е правилно форматиран

Форматирането представлява добавяне на символи, несъществени за компилатора, като интервали, табулации и нови редове, които структурират логически програмата и улесняват четенето й. Нека отново разгледаме кода на нашата първа програма (с краткия вариант за **`Main()`** метод):

```
class HelloCSharp
{
  static void Main()
  {
    System.Console.WriteLine("Hello C#!");
  }
}
```

Програмата съдържа седем реда и някои от редовете са повече или по-малко отместени навътре с помощта на табулации. Всичко това можеше да се напише и без отместване, например така:

```
class HelloCSharp
{
static void Main()
{
System.Console.WriteLine("Hello C#!");
}
}
```

или на един ред:

```
class HelloCSharp{static void Main(){System.Console.WriteLine(
"Hello C#!");}}
```

или дори така:

```
                                             class
    HelloCSharp
{
    static void                              Main()
    {                             System         .
Console.WriteLine("Hello C#!")       ;}          }
```

Горните примери ще се компилират и изпълнят по абсолютно същия начин като форматирания, но са далеч по-нечетливи, трудни за разбиране и осмисляне и съответно неудобни за промяна.

Не допускайте програмите ви да съдържат неформатиран код! Това силно намалява четимостта и довежда до трудно модифициране на кода.

Основни правила на форматирането

За да е форматиран кодът, трябва да следваме няколко важни правила за отместване:

- Методите се отместват по-навътре от дефиницията на класа;

- Съдържанието на методите се отмества по-навътре от дефиницията на метода;
- Отварящата фигурна скоба `{` трябва да е сама на ред и да е разположена точно под метода или класа, към който се отнася;
- Затварящата фигурна скоба `}` трябва да е сама на ред и да е поставена вертикално точно под съответната й отваряща скоба (със същото отместване като нея);
- Имената на класовете трябва да започват с главна буква;
- Имената на променливите трябва да започват с малка буква;
- Имената на методите трябва да започват с главна буква;

Имената на файловете съответстват на класовете

Всяка C# програма се състои от един или няколко класа. Прието е всеки клас да се дефинира в отделен файл с име, съвпадащо с името на класа и разширение `.cs`. При неизпълнение на тези изисквания програмата пак ще работи, но ориентацията в кода ще е затруднена. В нашия пример, тъй като класът се казва **`HelloCSharp`**, трябва да запишем неговият изходен (сорс) код във файл с име **`HelloCSharp.cs`**.

Езикът C# и платформата .NET

Първата версия на C# е разработена от Microsoft в периода 1999-2002 г. и е пусната официално в употреба през 2002 година, като част от .NET платформата, която има за цел да улесни съществено разработката на софтуер за Windows среда чрез качествено нов подход към програмирането, базиран на концепциите за "виртуална машина" и "управляван код". По това време езикът и платформата Java, изградени върху същите концепции, се радват на огромен успех във всички сфери на разработката на софтуер и разработката на C# и .NET е естественият отговор на Microsoft срещу успехите на Java технологията.

Езикът C#

C# е съвременен обектно-ориентиран език за програмиране от високо ниво с общо предназначение. Синтаксисът му е подобен на C и C++, но не поддържа много от неговите възможности с цел опростяване на езика, улесняване на програмирането и повишаване на сигурността.

Програмите на C# представляват един или няколко файла с разширение `.cs.`, в които се съдържат дефиниции на класове и други типове. Тези файлове се компилират от компилатора на C# (`csc`) до изпълним код и в резултат се получават асемблита – файлове със същото име, но с различно с разширение (`.exe` или `.dll`). Например, ако компилираме файла **`HelloCSharp.cs`**, ще получим като резултат файл с име **`HelloCSharp.exe`** (както и други помощни файлове, които не са от значение за момента).

Компилираният код може да се изпълни както всяка друга програма от нашия компютър (с двойно щракване върху нея). Ако се опитаме да изпълним компилирания C# код (например програмата **`HelloCSharp.exe`**) на компютър, на който няма .NET Framework, ще получим съобщение за грешка.

Ключови думи

Езикът C# използва следните **ключови думи** за построяване на своите програмни конструкции:

abstract	**event**	**new**	**struct**
as	**explicit**	**null**	**switch**
base	**extern**	**object**	**this**
bool	**false**	**operator**	**throw**
break	**finally**	**out**	**true**
byte	**fixed**	**override**	**try**
case	**float**	**params**	**typeof**
catch	**for**	**private**	**uint**
char	**foreach**	**protected**	**ulong**
checked	**goto**	**public**	**unchecked**
class	**if**	**readonly**	**unsafe**
const	**implicit**	**ref**	**ushort**
continue	**in**	**return**	**using**
decimal	**int**	**sbyte**	**virtual**
default	**interface**	**sealed**	**volatile**
delegate	**internal**	**short**	**void**
do	**is**	**sizeof**	**while**
double	**lock**	**stackalloc**	
else	**long**	**static**	
enum	**namespace**	**string**	

Още от създаването на първата версия на езика не всички ключови думи се използват. Някои от тях са добавени в по-късните версии. Основни конструкции в C# (които се дефинират и използват с помощта на ключовите думи) са класовете, методите, операторите, изразите, условните конструкции, циклите, типовете данни и изключенията.

Всички тези конструкции, както и употребата на повечето ключови думи от горната таблица, предстои да бъде разгледано подробно в следващите глави на настоящата книга.

Автоматично управление на паметта

Едно от най-големите предимства на .NET Framework е вграденото автоматично управление на паметта. То предпазва програмистите от сложната задача сами да заделят памет за обектите и да търсят подходящия момент за нейното освобождаване. Това сериозно повишава производителността на програмистите и увеличава качеството на програмите, писани на C#.

За управлението на паметта в .NET Framework се грижи специален компонент от CLR, наречен "**събирач на боклука**" или "**система за почистване на паметта**" (**garbage collector**). Основните задачи на събирача на боклука са да следи кога заделената памет за променливи и обекти вече не се използва, да я освобождава и да я прави достъпна за последващи заделяния на нови обекти.

> ⚠ **Важно е да се знае, че не е сигурно в точно кой момент паметта се изчиства от неизползваните обекти (например от локалните променливи). В спецификациите на езика C# е описано, че това става след като дадената променлива излезе от обхват, но не е посочено дали веднага или след изминаване на някакво време или при нужда от памет.**

Независимост от средата и от езика за програмиране

Едно от предимствата на .NET е, че програмистите, пишещи на различни .NET езици за програмиране могат да обменят кода си безпроблемно. Например C# програмист може да използва кода на програмист, написан на VB.NET, Managed C++ или F#. Това е възможно, тъй като програмите на различните .NET езици ползват обща система от типове данни и обща инфраструктура за изпълнение, както и единен формат на компилирания код (асемблита).

Като голямо предимство на .NET технологията се счита възможността веднъж написан и компилиран код да се изпълнява на различни операционни системи и хардуерни устройства. Можем да компилираме C# програма в Windows среда и да я изпълняваме както върху Windows така и върху Windows Mobile или Linux. Официално Microsoft поддържат .NET Framework само за Windows, Windows Mobile и Windows Phone платформи, но трети доставчици предлагат .NET имплементации за други операционни системи. Например проектът с отворен код Mono (www.mono-project.com) имплементира основната част от .NET Framework заедно с всички прилежащи библиотеки за Linux.

Common Intermediate Language (CIL)

Идеята за независимост от средата е заложена още при самото създаване на .NET платформата и се реализира с малка хитрина. Изходният код не се компилира до инструкции, предназначени за даден конкретен микропроцесор, и не използва специфични възможности на дадена операционна система, а се компилира до междинен език – така нареченият `Common Intermediate Language (CIL)`. Този език `CIL` не се изпълнява директно от микропроцесора, а се изпълнява от виртуална среда за изпълнения на CIL кода, наречена **Common Language Runtime (CLR)**.

Common Language Runtime (CLR) – сърцето на .NET

В самия център на .NET платформата работи нейното сърце – Common Language Runtime (CLR) – средата за контролирано изпълнение на управлявам код (CIL код). Тя осигурява изпълнение на .NET програми върху различни хардуерни платформи и операционни системи.

CLR е абстрактна изчислителна машина (виртуална машина). По аналогия на реалните електронноизчислителни машини тя поддържа набор от инструкции, регистри, достъп до паметта и входно-изходни операции. CLR осигурява контролирано изпълнение на .NET програмите, използвайки в пълнота възможностите на процесора и операционната система. CLR осъществява контролиран достъп до паметта и другите ресурси на машината като съобразява правата за достъп, зададени при изпълнението на програмата.

.NET платформата

.NET платформата, освен езика C#, съдържа в себе си CLR и множество помощни инструменти и библиотеки с готова функционалност. Съществуват няколко нейни разновидности съобразно целевата потребителска група:

- **.NET Framework** е най-използвания вариант на .NET среда, понеже тя е с общо, масово предназначение. Използва се при разработката на конзолни приложения, Windows програми с графичен интерфейс, уеб приложения и много други.

- **.NET Compact Framework** (CF) е "олекотена" версия на стандартния .NET Framework и се използва за разработка на приложения за мобилни телефони и други PDA устройства (използващи Windows Mobile Edition).

- **Silverlight** също е "олекотена" версия на .NET Framework, предназначена да се изпълнява в уеб браузърите за реализация на мултимедийни и RIA приложения (Rich Internet Applications).

.NET Framework

Стандартната версия на .NET платформата е предназначена за разработката и използването на конзолни приложения, настолни приложения, уеб приложения, уеб услуги, RIA приложения, и още много други. Почти всички .NET програмисти използват стандартната версия.

.NET технологиите

Въпреки своята големина и изчерпателност .NET платформата не дава инструменти за решаването на всички задачи от разработката на софтуер. Съществуват множество независими производители на софтуер, които разширяват и допълват стандартната функционалност, която предлага .NET Framework. Например фирми като българската софтуерна корпорация Telerik разработват допълнителни набори от компоненти за създаване на графичен потребителски интерфейс, средства за управление на уеб съдържание, библиотеки и инструменти за изготвяне на отчети и други инструменти за улесняване на разработката на приложения.

Разширенията, предлагани за .NET Framework, са програмни компоненти, достъпни за преизползване при писането на .NET програми. Преизползването на програмен код съществено улеснява и опростява разработката на софтуер, тъй като решава често срещани проблеми и предоставя наготово сложни алгоритми, имплементации на технологични стандарти и др. Съвременният програмист ежедневно използва готови библиотеки и така си спестява огромна част от усилията.

Да вземем за пример писането на програма, която визуализира данни под формата на графики и диаграми. Можем да вземем библиотека написана за .NET, която рисува самите графики. Всичко, от което се нуждаем, е да подадем правилните входни данни и библиотеката ще изрисува графиките вместо нас. Много е удобно и ефективно. Освен това води до понижаване на разходите за разработка, понеже програмистите няма да отделят време за разработване на допълнителната функционалност (в нашия случай самото чертаене на графиките, което е свързано със сложни математически изчисления и управление на видеокартата). Самото приложение също ще бъде с по-високо качество, понеже разширението, което се използва в него е разработвано и поддържано от специалисти, които имат доста повече опит в тази специфична област.

Повечето разширения се използват като инструменти, защото са сравнително прости. Съществуват и разширения, които представляват съвкупност от средства, библиотеки и инструменти за разработка, които имат сложна структура и вътрешни зависимости и е по-коректно да се нарекат **софтуерни технологии**. Съществуват множество .NET технологии с различни области на приложение. Типични примери са уеб технологиите (ASP.NET), позволяващи бързо и лесно да се пишат динамични уеб приложения и .NET RIA технологиите (Silverlight), които позволяват да се пишат мултимедийни приложения с богат потребителски интерфейс и работа в Интернет среда.

.NET Framework стандартно включва в себе си множество технологии и **библиотеки от класове** (class libraries) със стандартна функционалност, която програмистите ползват наготово в своите приложения. Например в системната библиотека има класове за работа с математически функции, изчисляване на логаритми и тригонометрични функции, които могат да се ползват наготово (класът `System.Math`). Друг пример е библиотеката за работа с мрежа (`System.Net`), която има готова функционалност за изпращане на e-mail (чрез класа `System.Net.Mail.MailMessage`) и за изтегляне на файл от Интернет (чрез класа `System.Net.WebClient`).

.NET технология наричаме съвкупността от .NET класове, библиотеки, инструменти, стандарти и други програмни средства и утвърдени подходи за разработка, които установяват технологична рамка при изграждането на определен тип приложения. **.NET библиотека** наричаме съвкупност от .NET класове, които предоставят наготово определен тип функционалност. Например .NET технология е ADO.NET, която предоставя стандартен подход за достъп до релационни бази от данни (като например Microsoft SQL Server и MySQL). Пример за библиотека са класовете от пакета `System.Data.SqlClient`, които предоставят връзка до SQL Server посредством технологията ADO.NET.

Някои технологии, разработвани от външни софтуерни доставчици, с времето започват да се използват масово и се утвърждават като технологични стандарти. Част от тях биват забелязани от Microsoft и биват включвани като разширения в следващите версии на .NET Framework. Така .NET платформата постоянно еволюира и се разширява с нови библиотеки и технологии. Например технологиите за обектно-релационна персистентност на данни (ORM технологиите) първоначално започнаха да се развиват като независими проекти и продукти (като проекта с отворен код NHibernate и продукта Telerik OpenAccess ORM), а по-късно набраха огромна популярност и доведоха до нуждата от вграждането им в .NET Framework. Така се родиха технологиите LINQ-to-SQL и ADO.NET Entity Framework съответно в .NET 3.5 и .NET 4.0.

Application Programming Interface (API)

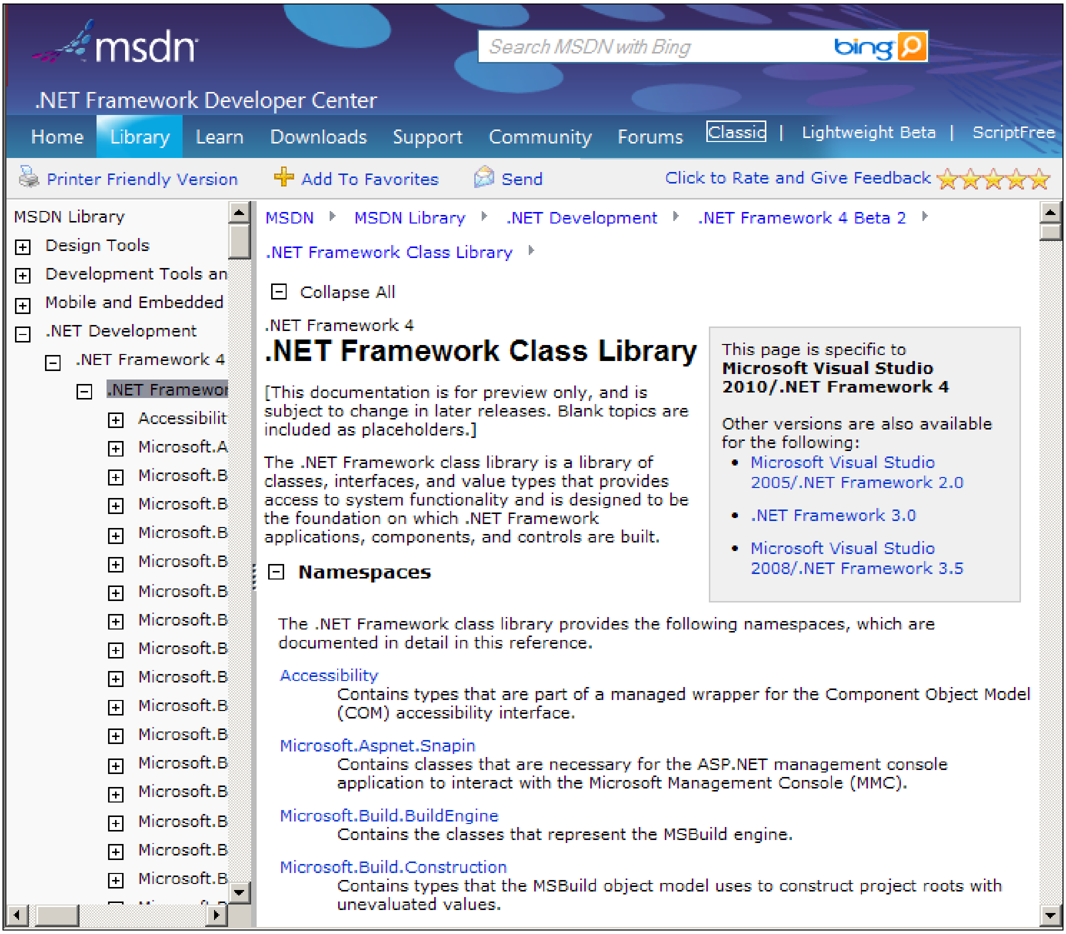
Всеки .NET инструмент или технология се използва, като се създават обекти и се викат техни методи. Наборът от публични класове и методи, които са достъпни за употреба от програмистите и се предоставят от технологиите, се нарича **Application Programming Interface** или просто **API**. За пример можем да дадем самия .NET API, който е набор от .NET библиотеки с класове, разширяващи възможностите на езика, добавяйки функционалност от високо ниво. Всички .NET технологии предоставят публичен API. Много често за самите технологии се говори просто като за API, предоставящ определена функционалност, като например API за работа с файлове, API за работа с графика, API за работа с принтер, уеб API и т.н. Голяма част от съвременния софтуер използва множество видове API, обособени като отделно ниво в софтуерните приложения.

.NET документацията

Много често се налага да се документира един API, защото той съдържа множество пространства от имена и класове. Класовете съдържат методи и параметри, смисълът на които не винаги е очевиден и трябва да бъде обяснен. Съществуват вътрешни зависимости между отделните класове и за правилната им употреба са необходими разяснения. Такива разяснения и технически инструкции за използване на дадена технология, библиотека или API и се наричат **документация**. Документацията представлява съвкупност от документи с техническо съдържание.

.NET Framework също има документация, разработвана и поддържана официално от Майкрософт. Тя е достъпна публично в Интернет и се разпространява заедно с .NET платформата като съвкупност от документи и инструменти за преглеждането им и за търсене.

Библиотеката **MSDN Library** представлява официалната документация на Microsoft за всички техни продукти за разработчици и софтуерни технологии. В частност документацията за .NET Framework е част от MSDN Library и е достъпна в Интернет на адрес: http://msdn.microsoft.com/en-us/library/ms229335(VS.100).aspx. Ето как изглежда тя:


msdn
Search MSDN with Bing
bing
.NET Framework Developer Center
Home
Library
Learn
Downloads
Support
Community
Forums
Classic | Lightweight Beta | ScriptFree
Printer Friendly Version
Add To Favorites
Send
Click to Rate and Give Feedback
MSDN Library
Design Tools
Development Tools an
Mobile and Embedded
.NET Development
.NET Framework 4
.NET Framewor
Accessibilit
Microsoft.A
Microsoft.B
Microsoft.B
Microsoft.B
Microsoft.B
Microsoft.B
Microsoft.B
Microsoft.B
Microsoft.B
Microsoft.B
Microsoft.B
Microsoft.B
Microsoft.B
Microsoft.B
Microsoft.B
Microsoft.B
MSDN ▸ MSDN Library ▸ .NET Development ▸ .NET Framework 4 Beta 2 ▸
.NET Framework Class Library ▸
Collapse All
.NET Framework 4

.NET Framework Class Library

[This documentation is for preview only, and is subject to change in later releases. Blank topics are included as placeholders.]

The .NET Framework class library is a library of classes, interfaces, and value types that provides access to system functionality and is designed to be the foundation on which .NET Framework applications, components, and controls are built.

This page is specific to **Microsoft Visual Studio 2010/.NET Framework 4**

Other versions are also available for the following:

- Microsoft Visual Studio 2005/.NET Framework 2.0
- .NET Framework 3.0
- Microsoft Visual Studio 2008/.NET Framework 3.5

Namespaces

The .NET Framework class library provides the following namespaces, which are documented in detail in this reference.

Accessibility
Contains types that are part of a managed wrapper for the Component Object Model (COM) accessibility interface.

Microsoft.Aspnet.Snapin
Contains classes that are necessary for the ASP.NET management console application to interact with the Microsoft Management Console (MMC).

Microsoft.Build.BuildEngine
Contains the classes that represent the MSBuild engine.

Microsoft.Build.Construction
Contains types that the MSBuild object model uses to construct project roots with unevaluated values.

Какво ви трябва, за да програмирате на C#?

След като разгледахме какво представляват .NET платформата, .NET библиотеките и .NET технологиите можем да преминем към писането, компилирането и изпълнението на C# програми.

Базовите изисквания, за да можете да програмирате на C# са инсталиран .NET Framework и текстов редактор. Текстовият редактор служи за създаване и редактиране на C# кода, а за компилиране и изпълнение се нуждаем от самия .NET Framework.

Текстов редактор

Текстовият редактор служи за писане на изходния код на програмата и за записването му във файл. След това кодът се компилира и изпълнява. Като текстов редактор можете да използвате вградения в Windows редактор Notepad (който е изключително примитивен и неудобен за работа) или да си изтеглите по-добър безплатен редактор като например някой от редакторите Notepad++ (http://notepad-plus.sourceforge.net) или PSPad (www.pspad.com).

Компилация и изпълнение на C# програми

Дойде време да компилираме и изпълним вече разгледания теоретично пример на проста програма, написана на C#. За целта трябва да направим следното:

- Да създадем файл с име `HelloCSharp.cs`;
- Да запишем примерната програма във файла;
- Да компилираме `HelloCSharp.cs` до файл `HelloCSharp.exe`;
- Да изпълним файла `HelloCSharp.exe`.

А сега, нека да го направим на компютъра!

Не забравяйте преди започването с примера, да инсталирате .NET Framework на компютъра си! В противен случай няма да можете да компилирате и да изпълните програмата.

По принцип .NET Framework се инсталира заедно с Windows, но в някои ситуации все пак би могъл да липсва. За да инсталирате .NET Framework на компютър, на който той не е инсталиран, трябва да го изтеглите от сайта на Майкрософт (http://download.microsoft.com). Ако не знаете коя версия да изтеглите, изберете последната версия.

Горните стъпки варират на различните операционни системи. Тъй като програмирането под Linux е малко встрани от фокуса на настоящата книга, ще разгледаме подробно какво ви е необходимо, за да напишете и изпълните примерната програма под Windows. За тези от вас които искат да се опитат да програмират на C# в Linux среда ще споменем необходимите инструменти и те ще имат възможност да си ги изтеглят и да експериментират.

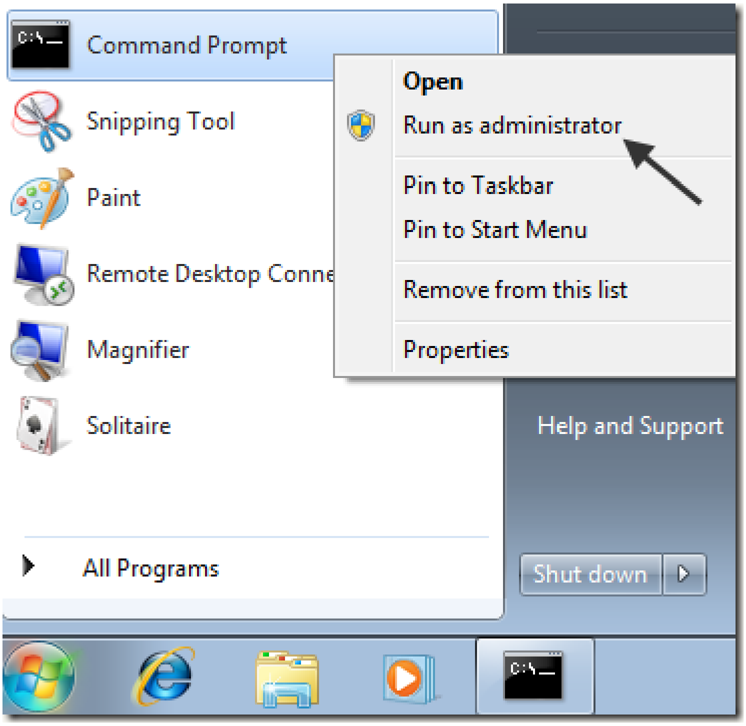
Ето го и кодът на нашата първа C# програма:

HelloCSharp.cs

```
class HelloCSharp
{
   static void Main()
   {
      System.Console.WriteLine("Hello C#!");
   }
}
```

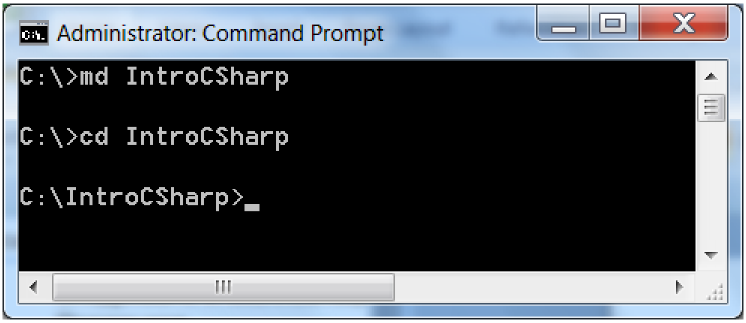
Компилиране на C# програми под Windows

Първо стартираме конзолата за команди на Windows, известна още като **Command Prompt** (това става от главното меню на Windows Explorer – Start -> Programs -> Accessories -> Command Prompt):


Command Prompt
Snipping Tool
Paint
Remote Desktop Conne
Magnifier
Solitaire
All Programs
Open
Run as administrator
Pin to Taskbar
Pin to Start Menu
Remove from this list
Properties
Help and Support
Shut down

За предпочитане е в конзолата да се работи с администраторски права, тъй като при липсата им някои операции не са позволени. Стартирането на **Command Prompt** с администраторски права става от контекстното меню, което се появява при натискане на десния бутон на мишката върху иконката на **Command Prompt** (вж. картинката).

Нека след това от конзолата създадем директория, в която ще експериментираме. Използваме командата `md` за създаване на директория и командата `cd` за влизане в нея:

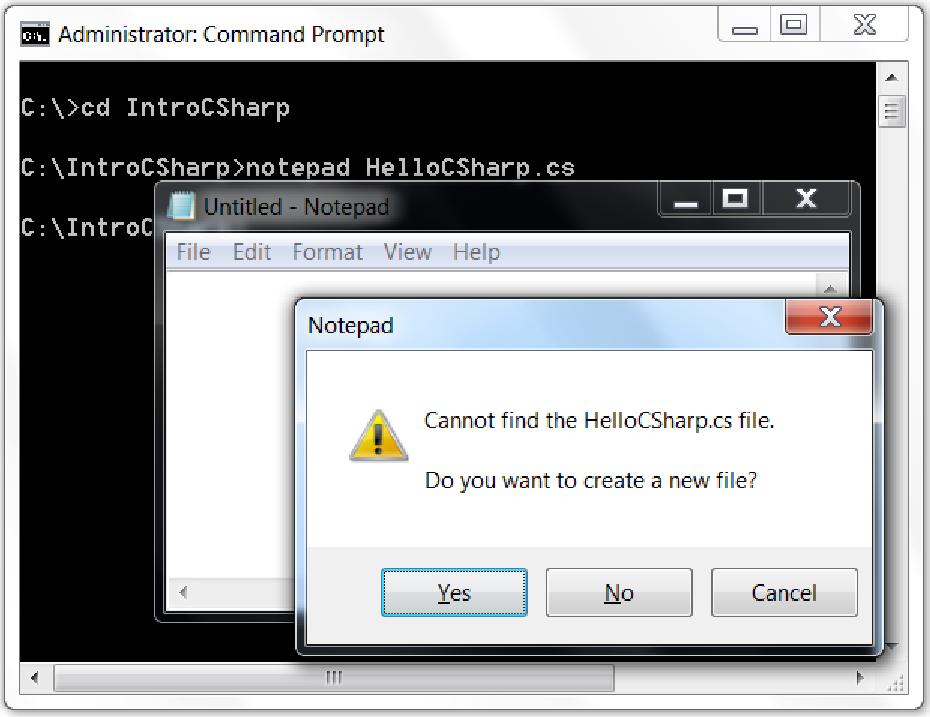

Administrator: Command Prompt

```
C:\>md IntroCSharp

C:\>cd IntroCSharp

C:\IntroCSharp>_
```

Директорията се казва **IntroCSharp** и се намира в **C:\.** Променяме текущата директория на **C:\IntroCSharp** и създаваме нов файл **HelloCSharp.cs,** като за целта използваме вградения в Windows текстов редактор Notepad. За да създадем файла, на конзолата изписваме следната команда:

```
notepad HelloCSharp.cs
```

Това стартира Notepad и той показва следния диалогов прозорец за създаване на несъществуващ файл:

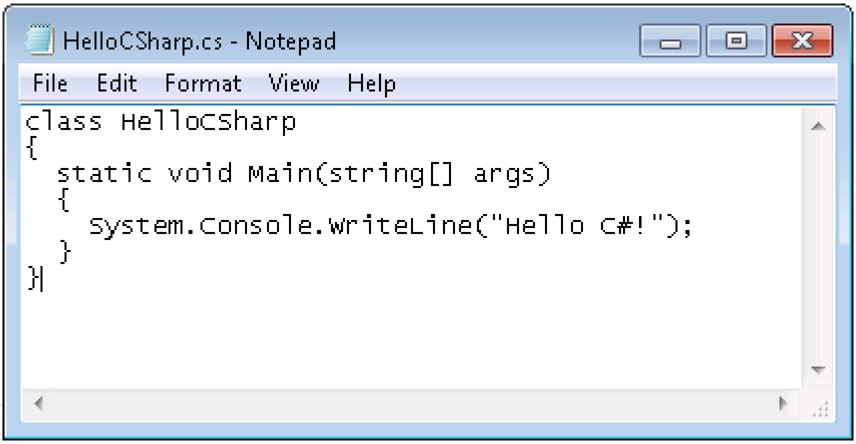

Administrator: Command Prompt
Untitled - Notepad
File Edit Format View Help
Notepad
Yes
No
Cancel

```
C:\>cd IntroCSharp

C:\IntroCSharp>notepad HelloCSharp.cs

C:\IntroC
```

Cannot find the HelloCSharp.cs file.

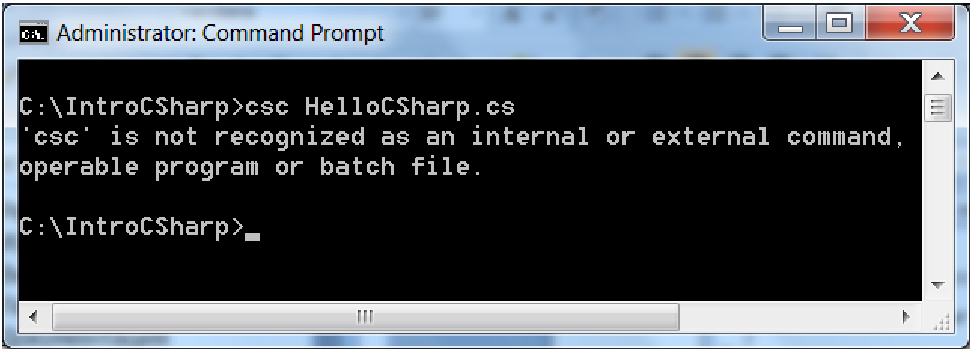
Do you want to create a new file?

Notepad ни пита дали искаме да бъде създаден нов файл, защото такъв в момента липсва. Отговаряме с "Yes". Следващата стъпка е да препишем програмата или просто да прехвърлим текста чрез копиране (Copy / Paste):



```
class HelloCSharp
{
  static void Main(string[] args)
  {
    System.Console.WriteLine("Hello C#!");
  }
}
```

Записваме чрез [Ctrl-S] и затваряме редактора Notepad с [Alt-F4]. Вече имаме изходния код на нашата примерна C# програма, записан във файла **`C:\IntroCSharp\HelloCSharp.cs`**. Остава да го компилираме и изпълним. Компилацията се извършва с компилатора **`csc.exe`**.

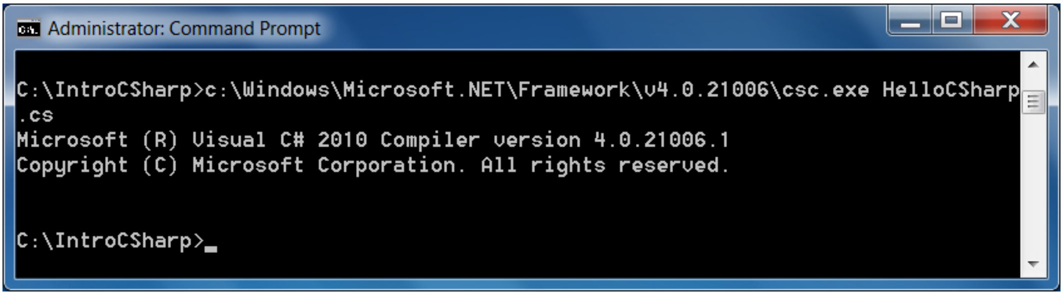


```
C:\IntroCSharp>csc HelloCSharp.cs
'csc' is not recognized as an internal or external command,
operable program or batch file.

C:\IntroCSharp>_
```

Ето, че получихме грешка – Windows не може да намери изпълним файл или вградена команда с име **`csc`**. Това е често срещан проблем и е нормално да се появи, ако сега започваме да работим с C#. Причините за него може да са следните:

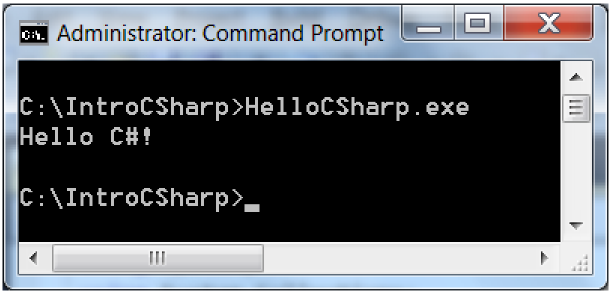
- Не е инсталиран .NET Framework;
- Инсталиран е .NET Framework, но директорията **`Microsoft.NET\Framework\v4.0`** не е в пътя за търсене на изпълними файлове и Windows не намира **`csc.exe`**, въпреки че той е наличен на диска.

Първият проблем се решава като се инсталира .NET Framework (в нашия случай – версия 4.0). Другият проблем може да се реши с промени в системния път (ще го направим след малко) или чрез използване на пълния път до **`csc.exe`**, както е показано на картинката долу. В нашия случай пълният път до C# компилатора на нашия твърд диск е следният: **`c:\Windows\Microsoft.NET\Framework\v4.0.21006\csc.exe`**. Да извикаме компилатора и да му подадем като параметър файла, който трябва да бъде компилиран (**`HelloCSharp.cs`**):



```
C:\IntroCSharp>c:\Windows\Microsoft.NET\Framework\v4.0.21006\csc.exe HelloCSharp
.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.21006.1
Copyright (C) Microsoft Corporation. All rights reserved.


C:\IntroCSharp>_
```

След изпълнението си **`csc`** приключва без грешки, като произвежда в резултат файла **`C:\IntroCSharp\HelloCSharp.exe`**. За да го изпълним, просто трябва да изпишем името му. Резултатът от изпълнението на нашата първа програма е съобщението "Hello, C#!", отпечатано на конзолата. Не е нещо велико, но е едно добро начало:

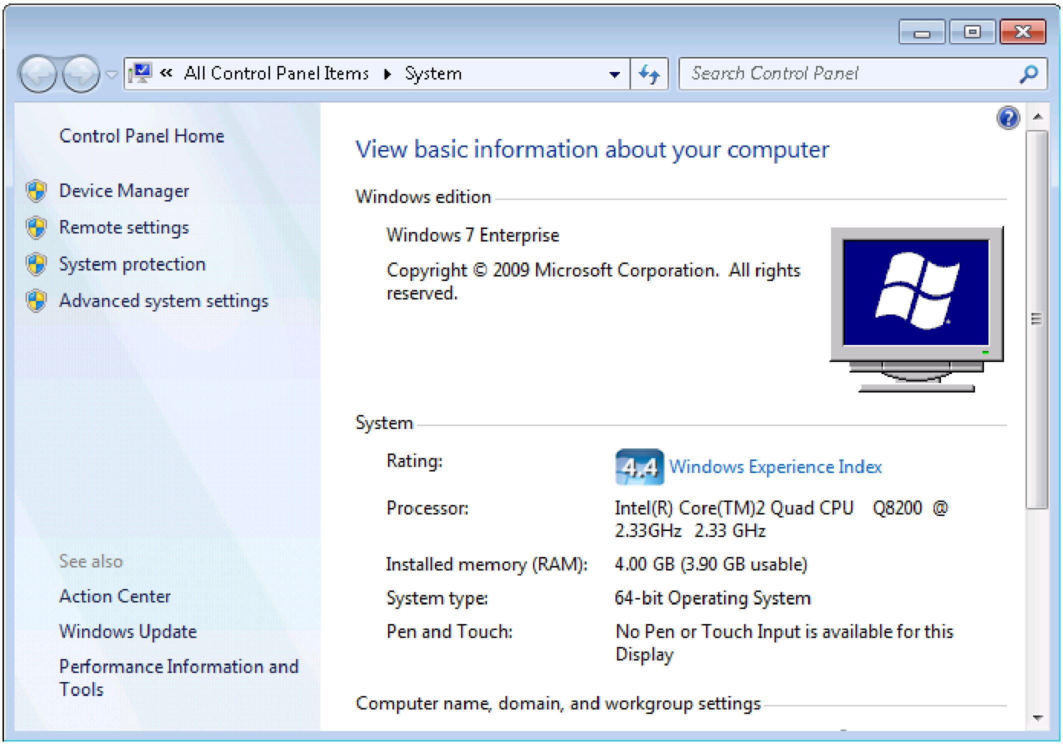


```
C:\IntroCSharp>HelloCSharp.exe
Hello C#!

C:\IntroCSharp>_
```

Промяна на системните пътища в Windows

Може би ви е досадно всеки път да изписвате пълния път до **`csc.exe`**, когато компилирате C# програми през конзолата. За да избегнете това, можете да редактирате системните пътища в Windows и след това да затворите конзолата и да я пуснете отново. Промяната на системните пътища в Windows става по следния начин:

1. Отиваме в контролния панел и избираме "System". Появява се следният добре познат прозорец:


« All Control Panel Items ▸ System
Search Control Panel
Control Panel Home
Device Manager
Remote settings
System protection
Advanced system settings
See also
Action Center
Windows Update
Performance Information and Tools
4,4
Windows Experience Index

View basic information about your computer

Windows edition

Windows 7 Enterprise

Copyright © 2009 Microsoft Corporation. All rights reserved.

System

Rating: 4,4 Windows Experience Index

Processor: Intel(R) Core(TM)2 Quad CPU Q8200 @ 2.33GHz 2.33 GHz

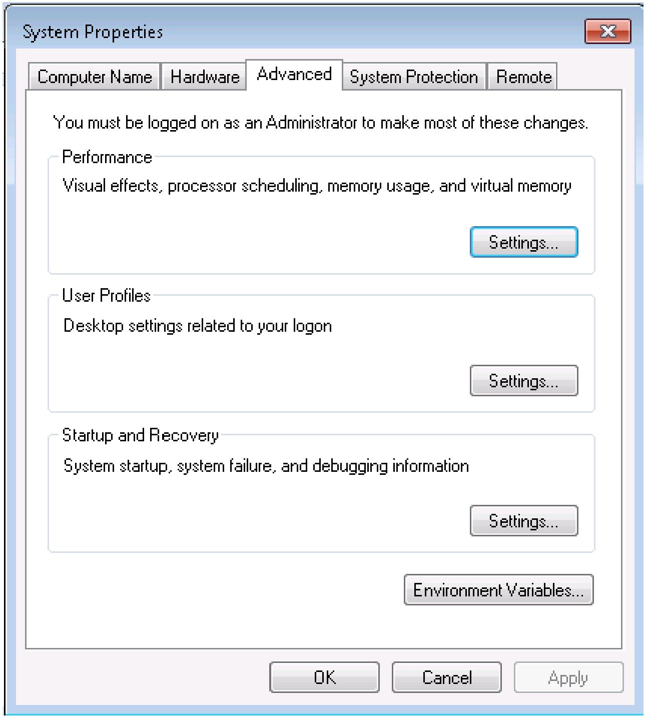
Installed memory (RAM): 4.00 GB (3.90 GB usable)

System type: 64-bit Operating System

Pen and Touch: No Pen or Touch Input is available for this Display

Computer name, domain, and workgroup settings

2. Избираме "Advanced System Settings". Появява се диалоговият прозорец "System Properties":


System Properties
Computer Name
Hardware
Advanced
System Protection
Remote
Settings...
Settings...
Settings...
Environment Variables...
OK
Cancel
Apply

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

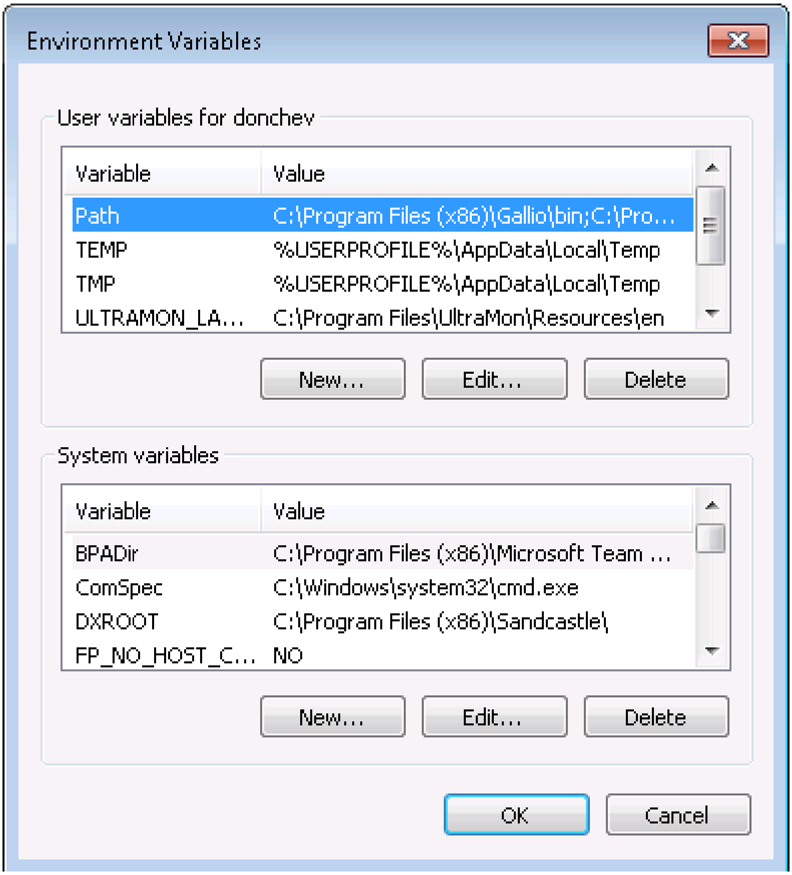
User Profiles

Desktop settings related to your logon

Startup and Recovery

System startup, system failure, and debugging information

3. Натискаме бутона "Environment Variables" и се показва прозорец с всички променливи на средата:


Environment Variables

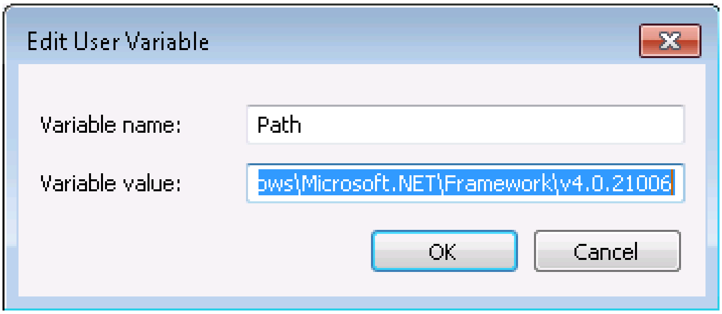
User variables for donchev

Variable	Value
Path	C:\Program Files (x86)\Gallio\bin;C:\Pro...
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp
ULTRAMON_LA...	C:\Program Files\UltraMon\Resources\en

System variables

Variable	Value
BPADir	C:\Program Files (x86)\Microsoft Team ...
ComSpec	C:\Windows\system32\cmd.exe
DXROOT	C:\Program Files (x86)\Sandcastle\
FP_NO_HOST_C...	NO

4. Избираме "Path" от списъка с променливите, както е показано на горната картинка и натискаме бутона "Edit". Появява се малко прозорче, в което добавяме пътя до директорията, където е инсталиран .NET Framework:


Edit User Variable

Variable name: Path

Variable value: ows\Microsoft.NET\Framework\v4.0.21006

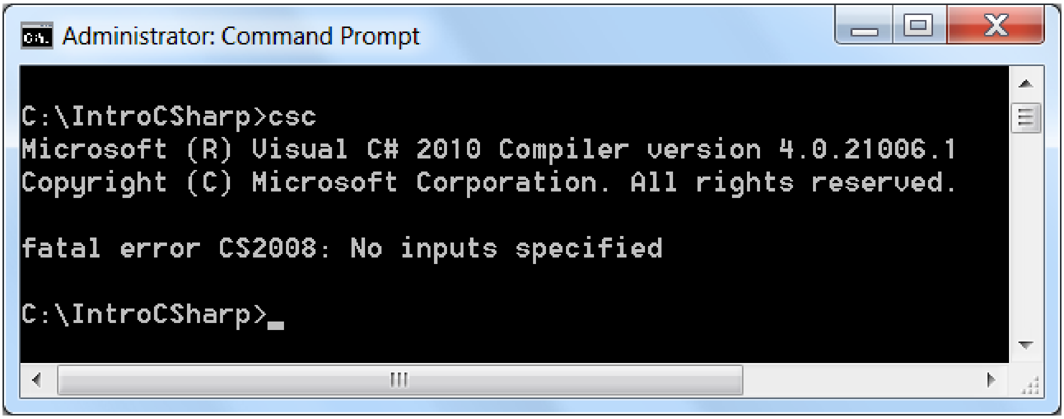
Разбира се, първо трябва да намерим къде е инсталиран .NET Framework. Стандартно той се намира някъде в директорията **`C:\Windows\Microsoft.NET`**, например в следната директория:

```
C:\Windows\Microsoft.NET\Framework\v.4.0.21006
```

Добавянето на допълнителен път към вече съществуващите пътища от променливата на средата **`Path`** се извършва като новият път се долепи до съществуващите и за разделител между тях се използва точка и запетая (`;`).

Бъдете внимателни, защото ако изтриете някой от вече съществуващите системни пътища, определени функции в Windows или част от инсталирания софтуер могат да спрат да функционират нормално!

5. Когато сме готови с пътя можем да се опитаме да извикаме **`csc.exe`**, без да посочваме пълния път до него. За целта отваряме **`cmd.exe`** (Command Prompt) и пишем командата "**`csc`**". Би трябвало да се изпише версията на C# компилатора и съобщение, че не е зададен входен файл:



```
C:\IntroCSharp>csc
Microsoft (R) Visual C# 2010 Compiler version 4.0.21006.1
Copyright (C) Microsoft Corporation. All rights reserved.

fatal error CS2008: No inputs specified

C:\IntroCSharp>_
```

Средата за разработка Visual Studio 2010 Express Edition

До момента разгледахме как се компилират и изпълняват C# програми през конзолата (Command Prompt). Разбира се, има и по-лесен начин – чрез използване на интегрирана среда за разработка, която може да изпълнява вместо нас всички команди, които използвахме. Нека разгледаме как се работи със среди за разработка и какво ни помагат те, за да си вършим по-лесно работата.

Интегрирани среди за разработка

В предходните примери разгледахме компилация и изпълнение на програма от един единствен файл. Обикновено програмите са съставени от много файлове, понякога дори десетки хиляди. Писането с текстов редактор, компилирането и изпълнението на една програма от командния ред е сравнително проста работа, но да направим това за голям проект, може да се окаже сложно и трудоемко занимание. За намаляване на сложността, улесняване на писането, компилирането и изпълнението на софтуерни приложения чрез един единствен инструмент съществуват визуални приложения, наречени **интегрирани среди за разработка** (Integrated Development Environment, IDE). Средите за разработка най-често предлагат множество допълнения към основните функции за разработка, като например дебъгване, изпълнение на unit тестове, проверка за често срещани грешки, достъп до хранилище за контрол на версиите и други.

Какво е Visual Studio 2010 Express Edition?

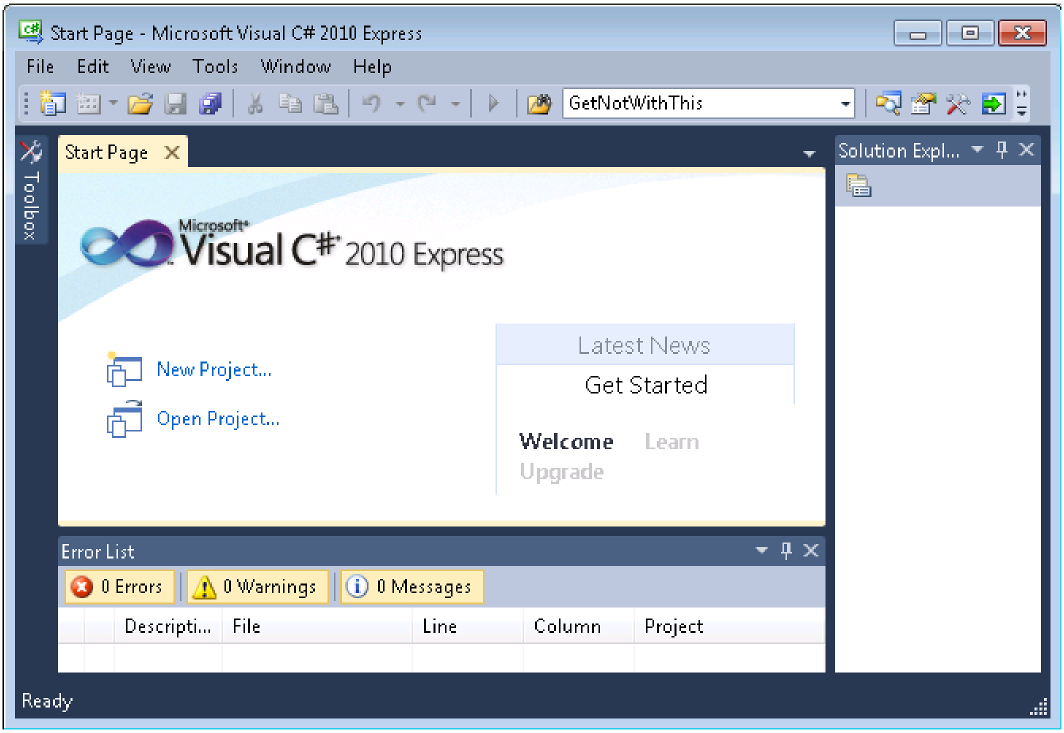
Visual Studio 2010 (VS 1020) е мощна интегрирана среда за разработка на софтуерни приложения за Windows и за платформата .NET Framework. VS 2010 поддържа различни езици за програмиране (например C#, VB.NET и C++) и различни технологии за разработка на софтуер (Win32, COM, ASP.NET, ADO.NET Entity Framework, Windows Forms, WPF, Silverlight и още десетки други Windows и .NET технологии). VS 2010 предоставя мощна интегрирана среда за писане на код, компилиране, изпълнение, дебъгване и тестване на приложения, дизайн на потребителски интерфейс (форми, диалози, уеб страници, визуални контроли и други), моделиране на данни, моделиране на класове, изпълнение на тестове, пакетиране на приложения и стотици други функции.

VS 2010 има безплатна версия наречена **Visual Studio 2010 Express Edition**, която може да се изтегли безплатно от сайта на Microsoft от адрес http://www.microsoft.com/express/.

В рамките на настоящата книга ще разгледаме само най-важните функции на VS 2010 Express – свързаните със самото програмиране. Това са функциите за създаване, редактиране, компилиране, изпълнение и дебъгване на програми.

Преди да преминем към примера, нека разгледаме малко по-подробно структурата на визуалния интерфейс на **Visual Studio 2010**. Основна съставна част са прозорците. Всеки прозорец реализира различна функция, свързана с разработката на приложения. Да разгледаме как изглежда **Visual Studio 2010** след начална инсталация и конфигурация по подразбиране. То съдържа няколко прозореца:

- **Start Page** – от началната страница можете лесно да отворите някой от последните си проекти или да стартирате нов, да направите първата си C# програма, да получите помощ за използването на C#.
- **Solution Explorer** – при незареден проект този прозорец е празен, но той ще стане част от живота ви като C# програмист. В него ще се показва структурата на проекта ви – всички файлове, от които се състои, независимо дали те са C# код, картинки, които ползвате, или някакъв друг вид код или ресурси.


Start Page - Microsoft Visual C# 2010 Express
File Edit View Tools Window Help
GetNotWithThis
Toolbox
Start Page
Solution Expl...
Microsoft Visual C# 2010 Express
New Project...
Open Project...
Latest News
Get Started
Welcome
Learn
Upgrade
Error List
0 Errors
0 Warnings
0 Messages
Descripti...
File
Line
Column
Project
Ready

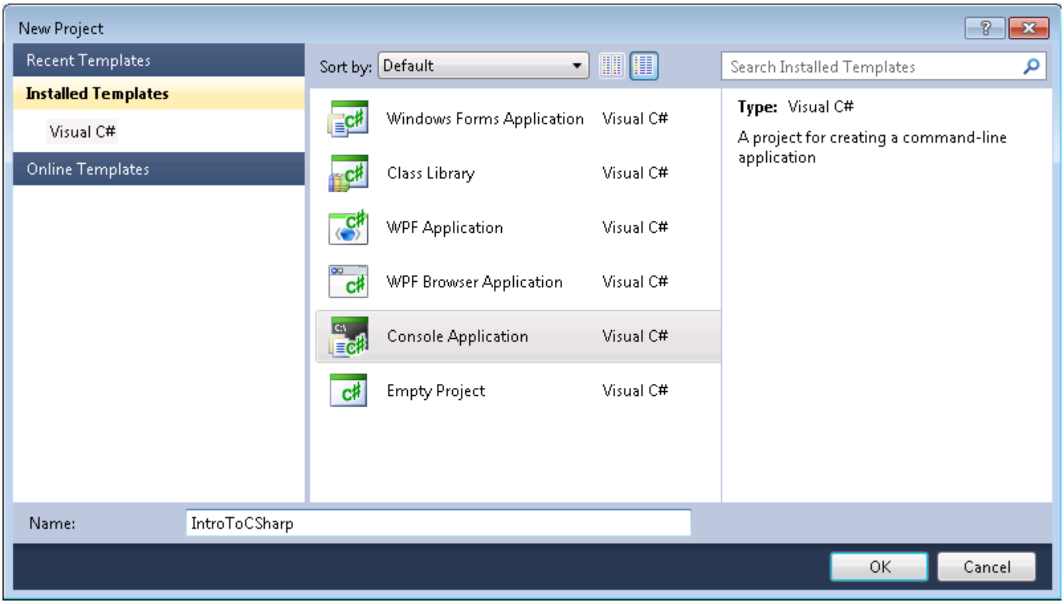
Съществуват още много други прозорци във Visual Studio с помощно предназначение, които няма да разглеждаме в момента.

Създаване на нов C# проект

Преди да направим каквото и да е във Visual Studio, трябва да създадем нов проект или да заредим съществуващ. Проектът логически групира множество файлове, предназначени да реализират някакво софтуерно приложение или система. За всяка програма е препоръчително да се създава отделен проект.

Проект във Visual Studio се създава чрез следване на следните стъпки:

- File -> New Project ...
- Появява се помощникът за нови проекти и в него са изброени типовете проекти, които можем да създадем. Имайте предвид, че понеже използваме безплатна версия на Visual Studio, предназначена главно за учащи, ще видим доста по-малко видове проекти, отколкото в стандартните, платени версии на VS:


New Project
Recent Templates
Installed Templates
Visual C#
Online Templates
Sort by: Default
Search Installed Templates
Windows Forms Application Visual C#
Class Library Visual C#
WPF Application Visual C#
WPF Browser Application Visual C#
Console Application Visual C#
Empty Project Visual C#
OK
Cancel

Type: Visual C#

A project for creating a command-line application

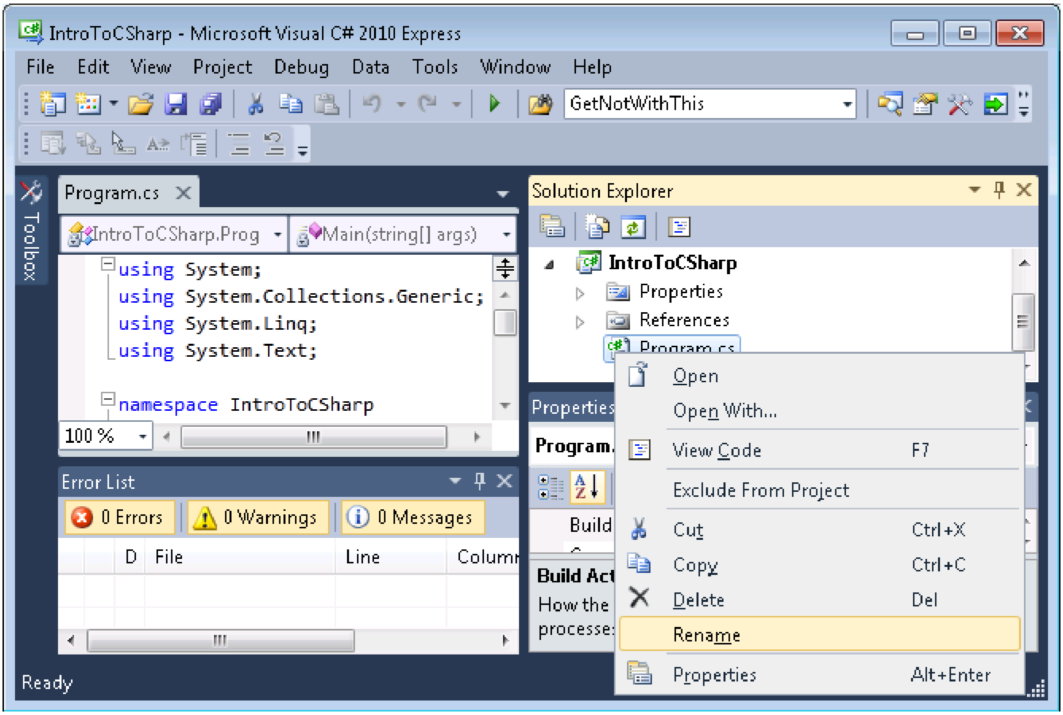
Name: IntroToCSharp

- Избираме **Console Application**. Конзолните приложения са програми, които ползват за вход и изход конзолата. Когато е необходимо да се въведат данни, те се въвеждат от клавиатурата, а когато се отпечатва нещо, то се появява в конзолата (т.е. като текст на екрана в прозореца на програмата). Освен конзолни приложенията могат да бъдат с графичен потребителски интерфейс (GUI), уеб приложения, уеб услуги, мобилни приложения, RIA приложения и други.

- В полето "Name" пишем името на проекта. В нашия случай избираме име **`IntroToCSharp`**.

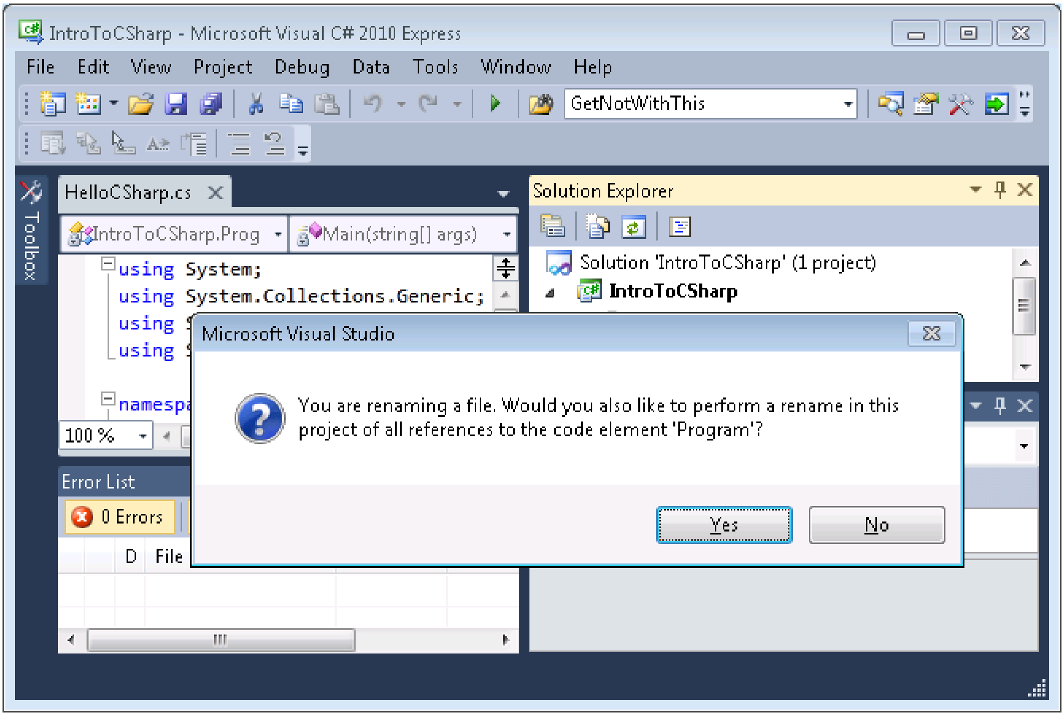
- Натискаме бутона [OK].

Новосъздаденият проект се показва в Solution Explorer. Автоматично е добавен и първият ни файл, съдържащ кода на програмата. Той носи името **`Program.cs`**. Важно е да задаваме смислени имена на нашите файлове, класове, методи и други елементи от програмата, за да можем след това лесно да ги намираме и да се ориентираме с кода. За да преименуваме файла **`Program.cs`**, щракваме с десен бутон върху него в Solution Explorer и избираме "Rename". Може да зададем за име на основния файл от нашата C# програма **`IntroToCSharp.cs`**. Преименуването а файл можем да изпълним и с клавиша [F2], когато е избран съответния файл от Solution Explorer:


IntroToCSharp - Microsoft Visual C# 2010 Express
File Edit View Project Debug Data Tools Window Help
GetNotWithThis
Toolbox
Program.cs
IntroToCSharp.Prog
Main(string[] args)
100 %
Error List
0 Errors
0 Warnings
0 Messages
D File Line Column
Ready
Solution Explorer
IntroToCSharp
Properties
References
Program.cs
Open
Open With...
View Code F7
Exclude From Project
Cut Ctrl+X
Copy Ctrl+C
Delete Del
Rename
Properties Alt+Enter
Properties
Program.
Build
Build Act
How the
processe

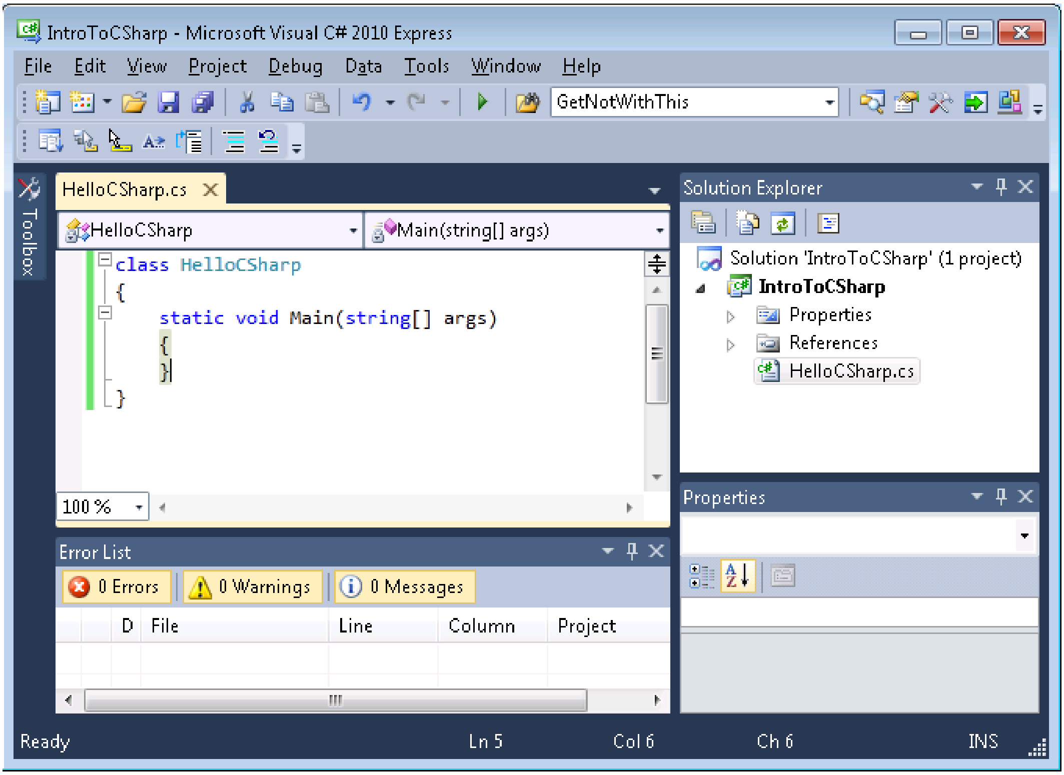
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace IntroToCSharp
```

Появява се диалогов прозорец, който ни пита дали искаме освен името на файла да преименуваме и името на класа. Избираме "Yes".


IntroToCSharp - Microsoft Visual C# 2010 Express
File Edit View Project Debug Data Tools Window Help
GetNotWithThis
Toolbox
HelloCSharp.cs
IntroToCSharp.Prog
Main(string[] args)
100 %
Error List
0 Errors
D File
Solution Explorer
Solution 'IntroToCSharp' (1 project)
IntroToCSharp
Microsoft Visual Studio
Yes
No

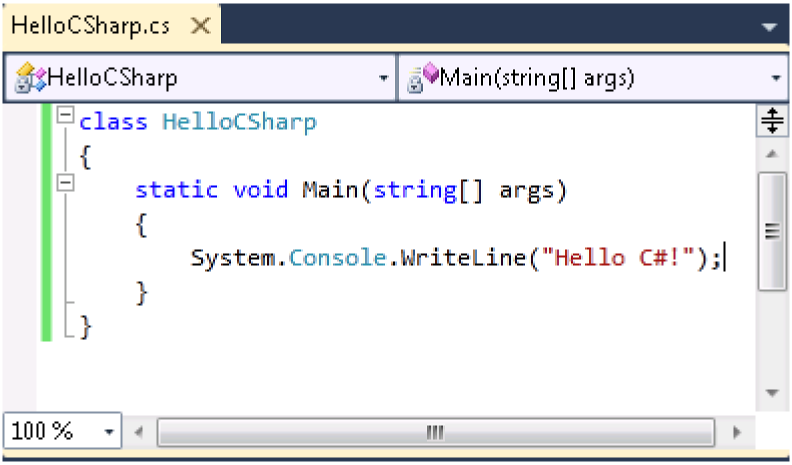
You are renaming a file. Would you also like to perform a rename in this project of all references to the code element 'Program'?

След като изпълним горните стъпки вече имаме първото конзолно приложение носещо името **IntroToCSharp** и съдържащо един единствен клас **HelloCSharp**:


IntroToCSharp - Microsoft Visual C# 2010 Express
File Edit View Project Debug Data Tools Window Help
GetNotWithThis
Toolbox
HelloCSharp.cs
HelloCSharp
Main(string[] args)
100 %
Solution Explorer
Solution 'IntroToCSharp' (1 project)
IntroToCSharp
Properties
References
HelloCSharp.cs
Properties
Error List
0 Errors
0 Warnings
0 Messages
D File Line Column Project
Ready
Ln 5
Col 6
Ch 6
INS

```
class HelloCSharp
{
    static void Main(string[] args)
    {
    }
}
```

Остава да допълним кода на метода **Main()**. По подразбиране кодът на **HelloCSharp.cs** би трябвало да е зареден за редактиране. Ако не е, щракваме два пъти върху файла **HelloCSharp.cs** в Solution Explorer прозореца, за да го заредим. Попълваме сорс кода:

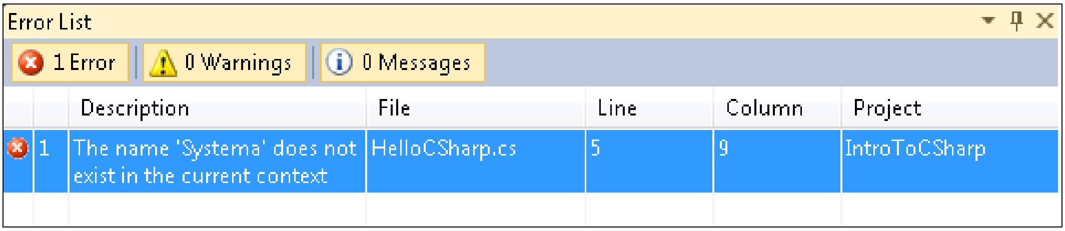

HelloCSharp.cs
HelloCSharp
Main(string[] args)
100 %

```
class HelloCSharp
{
    static void Main(string[] args)
    {
        System.Console.WriteLine("Hello C#!");
    }
}
```

Компилиране на сорс кода

Процесът на компилация във Visual Studio включва няколко стъпки:

- Проверка за синтактични грешки;



		Description	File	Line	Column	Project
✖	1	The name 'Systema' does not exist in the current context	HelloCSharp.cs	5	9	IntroToCSharp

- Проверка за други грешки, например липсващи библиотеки;
- Преобразуване на C# кода в изпълним файл (.NET асембли). При конзолни приложения се получава **`.exe`** файл.

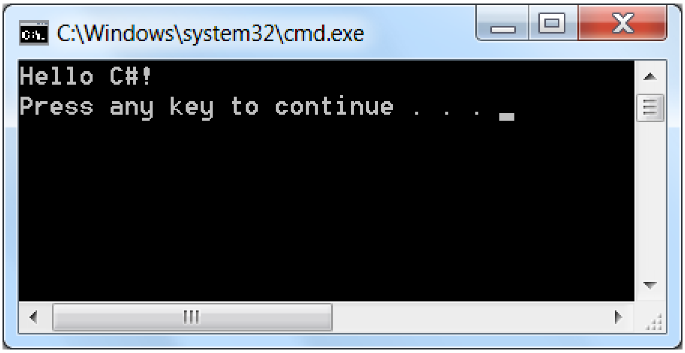
За да компилираме нашия примерен файл във Visual Studio натискаме клавиша [F6]. Обикновено още докато пишем и най-късно при компилация намерените грешки се подчертават в червено, за да привличат вниманието на програмиста, и се показват във визуализатора "Error List" (ако сте го изключили можете да го покажете от менюто "View" на Visual Studio).

Ако в проекта ни има поне една грешка, то тя се отбелязва с малък червен "x" в прозореца "Error List". За всяка грешка се визуализира кратко описание на проблема, име на файл, номер на ред и име на проект. Ако щракнем два пъти върху някоя от грешките в "Error List", Visual Studio ни прехвърля автоматично в съответния файл и на съответния ред в кода, на мястото в кода, където е възникнала грешката.

Стартиране на проекта

За да стартираме проекта натискаме [Ctrl+F5] (задържаме клавиша [Ctrl] натиснат и в това време натискаме клавиша [F5]).

Програмата се стартира и резултатът се изписва в конзолата, следван от текста "**`Press any key to continue . . .`**":



```
Hello C#!
Press any key to continue . . .
```

Последното съобщение не е част от резултата, произведен от програмата, а се показва от Visual Studio с цел да ни подсети, че програмата е завършила изпълнението си и да ни даде време да видим резултата. Ако стартираме програмата само с [F5], въпросното съобщение няма да се появи и резултатът ще изчезне веднага след като се е появил, защото програмата ще приключи и нейният прозорец ще бъде затворен. Затова използвайте [Ctrl+F5] за да стартирате своите конзолни програми.

Не всички типове проекти могат да се изпълняват. За да се изпълни C# проект, е необходимо той да съдържа точно един клас с `Main()` метод, деклариран по начина описан в началото на настоящата тема.

Дебъгване на програмата

Когато програмата ни съдържа грешки, известни още като **бъгове**, трябва да ги намерим и отстраним, т.е. да **дебъгнем** програмата. Процесът на дебъгване включва:

- Забелязване на проблемите (бъговете);
- Намиране на кода, който причинява проблемите;
- Оправяне на кода, така че програмата да работи правилно;
- Тестване, за да се убедим, че програмата работи правилно след нанесените корекции.

Процесът може да се повтори няколко пъти, докато програмата заработи правилно.

След като сме забелязали проблем в програмата, трябва да намерим кода, който го причинява. Visual Studio може да ни помогне с това, като ни позволи да проверим постъпково дали всичко работи, както е планирано.

За да спрем изпълнението на програмата на някакви определени места можем да поставяме **точки на прекъсване**, известни още като **стопери (breakpoints)**. Стоперът е асоцииран към ред от програмата. Програмата спира изпълнението си на тези редове, където има стопер и позволява постъпково изпълнение на останалите редове. На всяка стъпка може да проверяваме и дори да променяме стойностите на текущите променливи.

Дебъгването е един вид постъпково изпълнение на програмата на забавен кадър. То ни дава възможност по-лесно да вникнем в детайлите и да видим къде точно възникват грешките и каква е причината за тях.

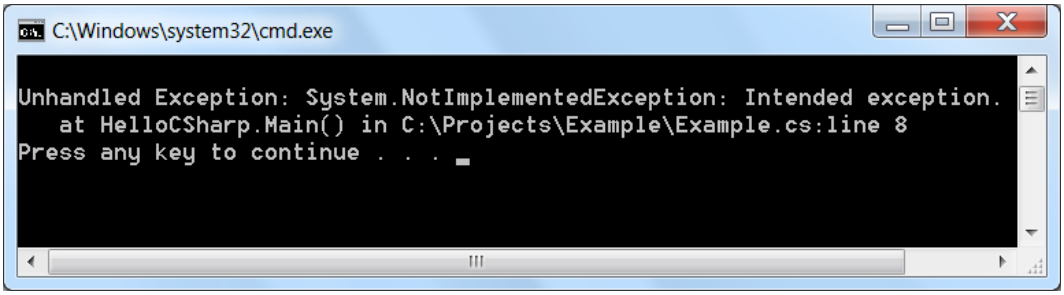
Нека направим грешка в нашата програма умишлено, за да видим как можем да се възползваме от стоперите. Ще добавим един ред в програмата, който ще създаде изключение по време на изпълнение (на изключенията ще се спрем подробно в главата "Обработка на изключения".

Засега нека направим програмата да изглежда по следния начин:

HelloCSharp.cs

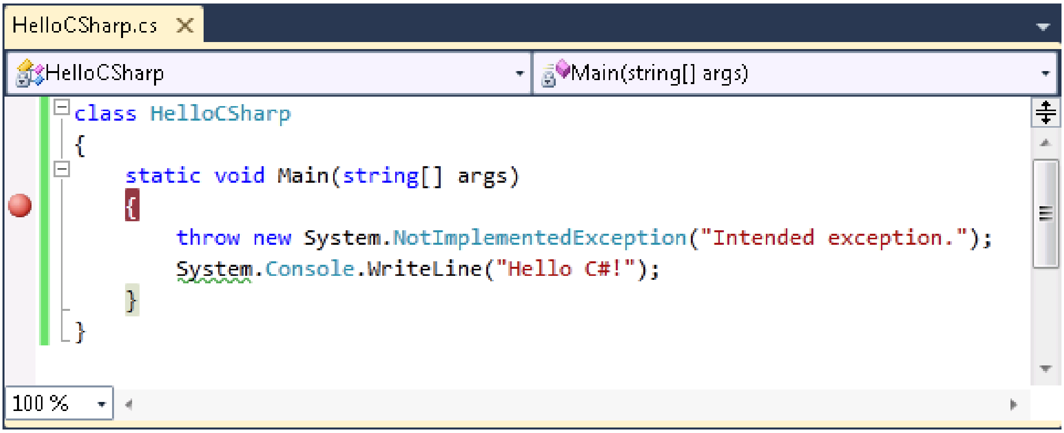
```
class HelloCSharp
{
   static void Main()
   {
      throw new System.NotImplementedException(
         "Intended exception.");
      System.Console.WriteLine("Hello C#!");
   }
}
```

Когато отново стартираме програмата с [Ctrl+F5] ще получим грешка и тя ще бъде отпечатана в конзолата:



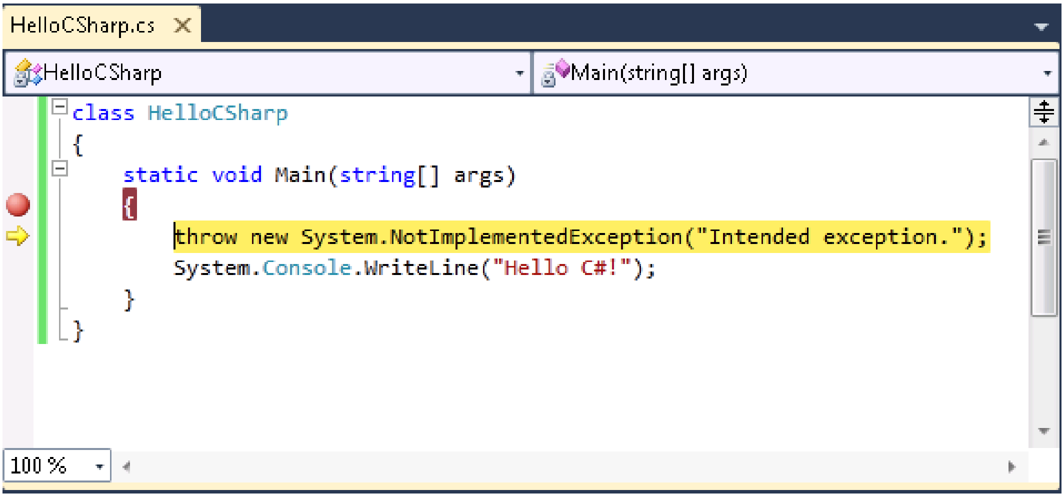
```
Unhandled Exception: System.NotImplementedException: Intended exception.
   at HelloCSharp.Main() in C:\Projects\Example\Example.cs:line 8
Press any key to continue . . .
```

Да видим как стоперите ще ни помогнат да намерим къде е проблемът. Преместваме курсора на реда, на който е отварящата скоба на класа и натискаме [F9] (така поставяме стопер на избрания ред). Появява се точка на прекъсване, където програмата ще спре изпълнението си, ако е стартирана в режим на дебъгване:



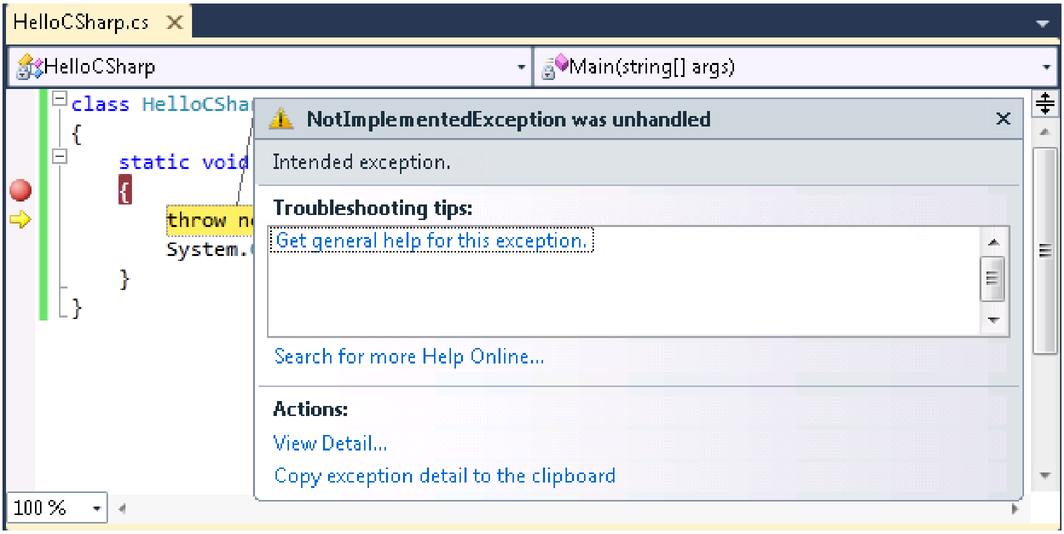
```
class HelloCSharp
{
    static void Main(string[] args)
    {
        throw new System.NotImplementedException("Intended exception.");
        System.Console.WriteLine("Hello C#!");
    }
}
```

Сега трябва да стартираме програмата в режим на отстраняване на грешки (в режим на дебъгване). Избираме Debug -> Start Debugging или натискаме [F5]. Програмата се стартира и веднага след това спира на първата точка на прекъсване, която срещне. Кодът се оцветява в жълто и можем да го изпълняваме постъпково. С клавиша [F10] преминаваме на следващия ред:


HelloCSharp.cs
HelloCSharp
Main(string[] args)
100 %

```
class HelloCSharp
{
    static void Main(string[] args)
    {
        throw new System.NotImplementedException("Intended exception.");
        System.Console.WriteLine("Hello C#!");
    }
}
```

Когато сме на даден ред и той е жълт, неговият код все още не е изпълнен. Изпълнява се след като го подминем. В случая все още не сме получили грешка, въпреки че сме на реда, който добавихме и който би трябвало да я предизвиква. Натискаме [F10] още веднъж, за да се изпълни текущият ред. Този път Visual Studio показва прозорец, който сочи реда, където е възникнала грешката, както и някои допълнителни детайли за нея:


HelloCSharp.cs
HelloCSharp
Main(string[] args)
Troubleshooting tips:
Get general help for this exception.
Search for more Help Online...
Actions:
View Detail...
Copy exception detail to the clipboard
100 %

NotImplementedException was unhandled

Intended exception.

След като вече знаем точно къде е проблемът в програмата, можем да го отстраним. За да стане това трябва първо да спрем изпълнението на програмата преди да е завършила. Избираме Debug -> Stop Debugging или натискаме [Shift + F5]. След това изтриваме проблемния ред и стартираме програмата в нормален режим (без проследяване) с [Ctrl+F5].

Алтернативи на Visual Studio

Както вече видяхме, въпреки че можем да минем и без Visual Studio на теория, това на практика не е добра идея. Работата по компилиране на един голям проект, отстраняването на грешки в кода и много други действия биха отнели много време извън Visual Studio.

От друга страна Visual Studio е платена среда за разработка на софтуер (в пълната му версия). Много хора трудно могат да си позволят да си закупят професионалните му версии (дори това може да е непосилно за малки фирми и отделни лица, които се занимават с програмиране).

Затова има някои алтернативи на Visual Studio (освен VS Express Edition), които са безплатни и могат да се справят нелошо със същите задачи.

SharpDevelop

Една от тях е SharpDevelop (#Develop). Можете да го намерите на следния сайт: http://www.icsharpcode.NET/OpenSource/SD/. #Develop е IDE за C# и се разработва като софтуер с отворен код. Той поддържа голяма част от функционалностите на Visual Studio 2010, но работи и под Linux и други операционни системи. Няма да го разглеждаме подробно, но го имайте предвид в случай, че ви е необходима среда за C# разработка и не можете да ползвате Visual Studio.

MonoDevelop

MonoDevelop е интегрирана среда за разработка на софтуер за .NET платформата. Той е напълно безплатен (с отворен код) и може да бъде свален от: http://monodevelop.com/. С MonoDevelop могат бързо и лесно да се пишат напълно функционални десктоп и ASP.NET приложения за Linux, Mac OSX и Windows. С него програмистите могат лесно да прехвърлят проекти, създадени с Visual Studio, към Mono платформата и да ги направят напълно функциониращи под други платформи.

Декомпилиране на код

Понякога на програмистите им се налага да видят кода на даден модул или програма, които не са писани от тях самите и за които не е наличен сорс код. Процесът на генерирането на сорс код от съществуващ изпълним бинарен файл (.NET асембли – `.exe` или `.dll`) се нарича **декомпилация**.

Декомпилацията на код може да ви се наложи в следните случаи:

- Искате да видите как е реализиран даден алгоритъм, за който знаете, че работи добре.
- Имате няколко варианта, когато използвате нечия библиотека и искате да изберете оптималния.

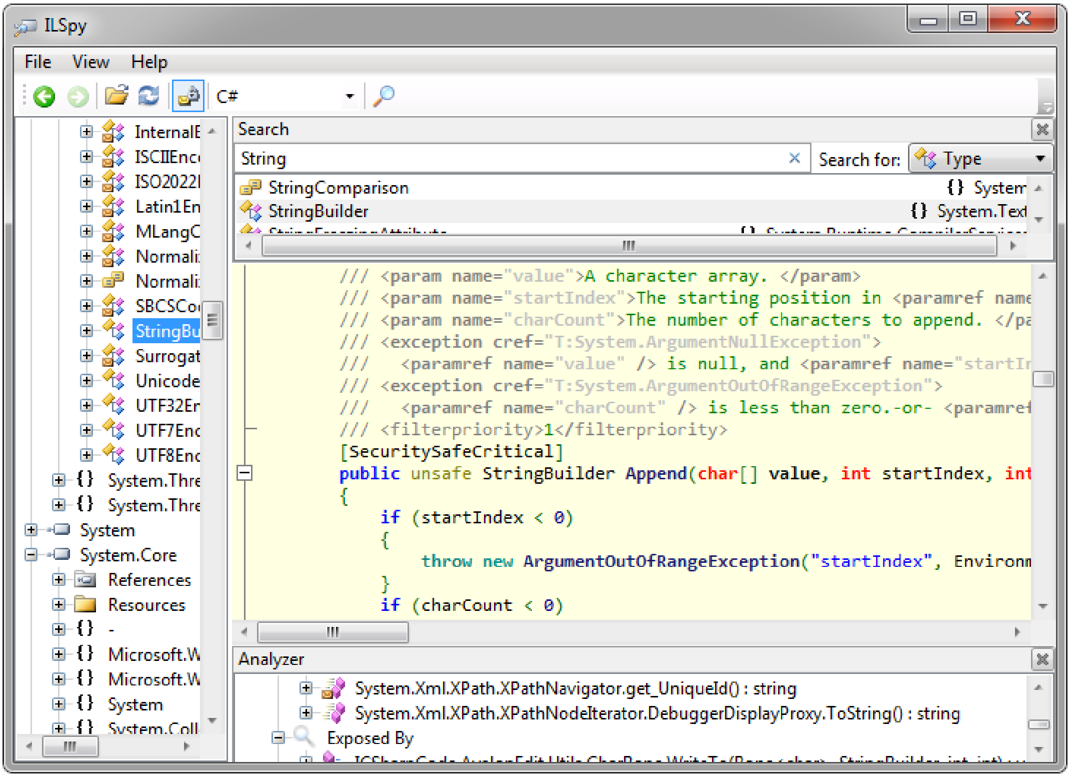
- Нямате информация как работи дадена библиотека, но имате компилиран код (асембли), който я използва и искате да разберете как точно го прави.

Декомпилацията се извършва с помощни инструменти, които не са част от Visual Studio. Най-използваният такъв инструмент беше Red Gate's Reflector (преди да стане платен в началото на 2011). В момента фирма Telerik се занимава с разработката на декомпилатор, който може да бъде изтеглен безплатно от сайта на Telerik на следния адрес: http://www.telerik.com/products/decompiling.aspx. Той се казва **JustDecompile** и се интегрира с Visual Studio. В момента на писане на тази книга, този декомпилатор е още в бета версия. Единственото условие да свалите декомпилатора е да се регистрирате в сайта на Telerik.

Друг много добър инструмент за декомпилация е програмата **ILSpy**, разработвана от хората от IC#Code, които разработват и алтернатива на Visual Studio, която пък се казва SharpDevelop. ILSpy може да бъде свален от: http://wiki.sharpdevelop.net/ilspy.ashx

Програмата няма нужда от инсталация. След като бъде стартирана ILSpy зарежда някои от стандартните библиотеки от .NET Framework. Чрез менюто File -> Open, можете да отворите избрано от вас .NET асембли. Можете да заредите и асембли от GAC (Global Assembly Cache).

Ето как изглежда програмата по време на работа:


ILSpy
File
View
Help
C#
InternalE
ISCIIEnc
ISO2022I
Latin1En
MLangC
Normali
Normali
SBCSCo
StringBu
Surrogat
Unicode
UTF32Er
UTF7Enc
UTF8Enc
{} System.Thre
{} System.Thre
System
System.Core
References
Resources
{} -
{} Microsoft.W
{} Microsoft.W
{} System
{} System.Coll
Search
Search for:
Type
Analyzer
Exposed By

String

StringComparison {} System
StringBuilder {} System.Text

```
/// <param name="value">A character array. </param>
/// <param name="startIndex">The starting position in <paramref name
/// <param name="charCount">The number of characters to append. </pa
/// <exception cref="T:System.ArgumentNullException">
///   <paramref name="value" /> is null, and <paramref name="startIn
/// <exception cref="T:System.ArgumentOutOfRangeException">
///   <paramref name="charCount" /> is less than zero.-or- <paramref
/// <filterpriority>1</filterpriority>
[SecuritySafeCritical]
public unsafe StringBuilder Append(char[] value, int startIndex, int
{
    if (startIndex < 0)
    {
        throw new ArgumentOutOfRangeException("startIndex", Environm
    }
    if (charCount < 0)
```

System.Xml.XPath.XPathNavigator.get_UniqueId() : string
System.Xml.XPath.XPathNodeIterator.DebuggerDisplayProxy.ToString() : string

По два начина можете да видите как е реализиран даден метод. Ако искате да видите примерно как работи статичния метод **`System.Currency.ToDecimal`** първо можете да използвате дървото вляво и да намерите класа **`Currency`** в пространството от имена **`System`**, след това да изберете метода **`ToDecimal`**. Достатъчно е да натиснете върху даден метод, за да видите неговия C# код. Друг вариант да намерите даден клас е чрез търсене с търсачката на ILSpy. Тя търси в имената на всички класове, интерфейси, методи, свойства и т.н. от заредените в програмата асемблита. За съжаление във версията към момента на писането на книгата (ILSpy 1.0 beta) програмата поддържа декомпилиране само до езиците C# и IL.

JustDecompiler и ILSpy са изключително полезни инструменти, който се използват почти ежедневно при разработката на .NET софтуер и затова задължително трябва да си изтеглите поне едно от тях и да си поиграете с него. Винаги, когато се чудите как работи даден метод или как е имплементирано дадено нещо в някое асембли, можете да разчитате на декомпилатора, за да научите.

C# под Linux

В момента в България (а може би и на световно ниво) програмирането на C# за Linux е доста по-слабо развито от това за Windows. Въпреки всичко не искаме да го пропуснем с лека ръка и затова ще ви дадем отправни точки, от които можете да тръгнете сами, ако ползвате Linux.

Програмиране на C# под Linux?

Най-важното, което ни трябва, за да програмираме на C# под Linux е имплементация на .NET Framework. Microsoft .NET Framework не се поддържа за Linux, но има друга .NET имплементация, която се нарича Mono. Можете да си изтеглите Mono (който се разработва и разпространява като свободен софтуер) от неговия официален уеб сайт: http://www.go-mono.com/mono-downloads/download.html. Mono позволява да компилираме и изпълняваме програми на C# в Linux среда и върху други операционни системи. Той съдържа C# компилатор, CLR, garbage collector, стандартните .NET библиотеки и всичко останало, което имаме в Microsoft .NET Framework под Windows.

Разбира се, Visual Studio за Linux също няма, но можем да използваме аналога на #Develop – monoDevelop. Него можете да изтеглите от: http://www.monodevelop.com.

Упражнения

1. Запознайте се с Microsoft Visual Studio, Microsoft Developer Network (MSDN) Library Documentation. Инсталирайте
2. Да се намери описанието на класа **`System.Console`** в стандартната .NET API документация (MSDN Library).

3. Да се намери описанието на метода **`System.Console.WriteLine()`** с различните негови възможни параметри в MSDN Library.
4. Да се компилира и изпълни примерната програма от примерите в тази глава през командния ред (конзолата) и с помощта на Visual Studio.
5. Да се модифицира примерната програма, така че да изписва различно поздравление, например "Добър ден!".
6. Напишете програма, която изписва вашето име и фамилия на конзолата.
7. Напишете програма, която извежда на конзолата числата 1, 101, 1001 на нов ред.
8. Напишете програма, която извежда на конзолата текущата дата и час.
9. Напишете програма, която извежда корен квадратен от числото 12345.
10. Напишете програма, която извежда първите 100 члена на редицата 2, -3, 4, -5, 6, -7, 8.
11. Направете програма, която прочита от конзолата вашата възраст и изписва (също на конзолата) каква ще бъде вашата възраст след 10 години.
12. Опишете разликите между C# и .NET Framework.
13. Направете списък с най популярните програмни езици. С какво те се различават от C#?
14. Да се декомпилира примерната програма от задача 5.

Решения и упътвания

1. Ако разполагате с DreamSpark акаунт или вашето училище или университет предлага безплатен достъп до продуктите на Microsoft, си инсталирайте пълната версия на Microsoft Visual Studio. Ако нямате възможност да работите с пълната версия на Microsoft Visual Studio, можете безплатно да си изтеглите Visual C# Express от сайта на Microsoft, който е напълно безплатен за използване с учебна цел.
2. Използвайте адреса, даден в раздела .NET документация към тази глава. Отворете го и търсете в йерархията вляво. Може да направите и търсене в Google – това също работи добре и често пъти е най-бързият начин да намерим документацията за даден .NET клас.
3. Използвайте същия подход като в предходната задача.
4. Следвайте инструкциите от раздела Компилация и изпълнение на C# програми.
5. Използвайте кода на примерната C# програма от тази глава и променете съобщението, което се отпечатва. Ако имате проблеми с

кирилицата, сменете т. нар. System Locale с български от прозореца "Region and Language" в контролния панел на Windows.

6. Потърсете как се използва метода **`System.Console.Write()`**.
7. Използвайте метода **`System.Console.WriteLine()`**.
8. Потърсете какви възможности предлага класа **`System.DateTime`**.
9. Потърсете какви възможности предлага класа **`System.Math`**.
10. Опитайте се сами да научите от интернет как се ползват цикли в C#.
11. Използвайте методите **`System.Console.ReadLine()`**, **`int.Parse()`** и **`System.DateTime.AddYears`**().
12. Направете проучване в интернет и се запознайте детайлно с разликите между тях.
13. Проучете най-популярните езици и вижте примерни програми на тях. Сравнете ги с езика C#.
14. Първо изтеглете и инсталирайте JustDecompile или ILSpy (повече информация за тях можете да намерите в тази глава). След като ги стартирате, отворете компилирания файл, от вашата програма. Той се намира в поддиректория **`bin\Debug`** на вашия проект. Например, ако вашият проект се казва **`TestCSharp`** и се намира в **`C:\Projects`**, то компилираното асембли на вашата програма ще е файлът **`C:\Projects\TestCSharp\bin\Debug\TestCSharp.exe`**.

Telerik Academy
SCHOOL

Стани част от Академията на Телерик за ученици!

“Основната цел на обучението е да израстнеш, а нашите умове, за разлика от телата ни, могат да растат през целия ни живот!“ Мортимър Адлър

АКАДЕМИЯТА НА ТЕЛЕРИК за ученици предоставя съвременно, специализирано обучение по софтуерно инженерство за ученици от средните училища в България. Нашата основна цел е да помогнем на младите момчета и момичета да се подготвят за Националната олимпиада по информационни технологии (НОИТ) и да придобият солидни теоретични и практически умения за разработка на софтуер и софтуерни технологии.

В Училищната академия на Телерик по софтуерно инженерство:

- ➔ **Стимулираме креативността и мисленето на учениците**
- ➔ **Всеки ученик получава индивидуално внимание и съдействие**
- ➔ **Полагаме основите за писане на качествен програмен код**
- ➔ **Изучаваме съвременните софтуерни технологии**
- ➔ **Подготвяме учениците за Националната олимпиада по информационни технологии**
- ➔ **Предлагаме голямо разнообразие от практически и теоретични обучения**

Програмата на Академията включва:

C#, .NET технологии, бази данни, SQL, Silverlight, 3D графики и моделиране, разработка на игри, програмиране на микроконтролери, WPF, ASP.NET, HTML5, разработка на мобилни приложения, основи на софтуерното инженерство, работа в екип

schoolacademy.telerik.com

groups.google.com/group/it-olymp   *facebook.com/TelerikSchoolAcademy*

telerik
deliver more than expected

Глава 2. Примитивни типове и променливи

В тази тема...

В настоящата тема ще разгледаме примитивните типове и променливи в C# – какво представляват и как се работи с тях. Първо ще се спрем на типовете данни – целочислени типове, реални типове с плаваща запетая, булев тип, символен тип, стринг и обектен тип. Ще продължим с променливите, какви са техните характеристики, как се декларират, как им се присвоява стойност и какво е инициализация на променлива. Ще се запознаем и с типовете данни в C# – стойностни и референтни. Накрая ще разгледаме различните видове литерали и тяхното приложение.

Какво е променлива?

Една типична програма използва различни стойности, които се променят по време на нейното изпълнение. Например създаваме програма, която извършва някакви пресмятания върху стойности, които потребителят въвежда. Стойностите, въведени от даден потребител, ще бъдат очевидно различни от тези, въведени от друг потребител. Това означава, че когато създава програмата, програмистът не знае всички възможни стойности, които ще бъдат въвеждани като вход, а това налага да се обработват всички различни стойности, въвеждани от различните потребители.

Когато потребителят въведе нова стойност, която ще участва в процеса на пресмятане, можем да я съхраним (временно) в оперативната памет на нашия компютър. Стойностите в тази част на паметта се променят постоянно и това е довело до наименованието им – променливи.

Типове данни

Типовете данни представляват множества (диапазони) от стойности, които имат еднакви характеристики. Например типът **`byte`** задава множеството от цели числа в диапазона [0….255].

Характеристики

Типовете данни се характеризират с:

- Име – например **`int`**;
- Размер (колко памет заемат) – например 4 байта;
- Стойност по подразбиране (**default value**) – например 0.

Видове

Базовите типове данни в C# се разделят на следните видове:

- Целочислени типове – **`sbyte`**, **`byte`**, **`short`**, **`ushort`**, **`int`**, **`uint`**, **`long`**, **`ulong`**;
- Реални типове с плаваща запетая – **`float`**, **`double`**;
- Реални типове с десетична точност – **`decimal`**;
- Булев тип – **`bool`**;
- Символен тип – **`char`**;
- Символен низ (стринг) – **`string`**;
- Обектен тип – **`object`**.

Тези типове данни се наричат **примитивни (built-in types)**, тъй като са вградени в езика C# на най-ниско ниво. В таблицата по-долу можем да видим изброените по-горе типове данни, техният обхват и стойностите им по подразбиране:

Тип данни	Стойност по подразбиране	Минимална стойност	Максимална стойност
`sbyte`	0	-128	127
`byte`	0	0	255
`short`	0	-32768	32767
`ushort`	0	0	65535
`int`	0	-2147483648	2147483647
`uint`	0u	0	4294967295
`long`	0L	-9223372036854775808	9223372036854775807
`ulong`	0u	0	18446744073709551615
`float`	0.0f	$\pm 1.5\times 10^{-45}$	$\pm 3.4\times 10^{38}$
`double`	0.0d	$\pm 5.0\times 10^{-324}$	$\pm 1.7\times 10^{308}$
`decimal`	0.0m	$\pm 1.0\times 10^{-28}$	$\pm 7.9\times 10^{28}$
`boolean`	false	Възможните стойности са две – **`true`** или **`false`**	
`char`	'\u0000'	'\u0000'	'\uffff'
`object`	null	-	-
`string`	null	-	-

Съответствие на типовете в C# и в .NET Framework

Примитивните типове данни в C# имат директно съответствие с типове от общата система от типове (CTS) от .NET Framework. Например типът **`int`** в C# съответства на типа **`System.Int32`** от CTS и на типа **`Integer`** в езика VB.NET, а типът **`long`** в C# съответства на типа **`System.Int64`** от CTS и на типа **`Long`** в езика VB.NET. Благодарение на общата система на типовете (CTS) в .NET Framework има съвместимост между различните езици за програмиране (като например C#, Managed C++, VB.NET и F#). По същата причина типовете **`int`**, **`Int32`** и **`System.Int32`** в C# са всъщност различни псевдоними за един и същ тип данни – 32 битово цяло число със знак.

Целочислени типове

Целочислените типове отразяват целите числа и биват **`sbyte`**, **`byte`**, **`short`**, **`ushort`**, **`int`**, **`uint`**, **`long`** и **`ulong`**. Нека ги разгледаме един по един.

Типът **`sbyte`** е 8-битов целочислен тип **със знак** (**signed integer**). Това означава, че броят на възможните му стойности е 2^8, т.е. 256 възможни стойности общо, като те могат да бъдат както положителни, така и отрицателни. Минималната стойност, която може да се съхранява в `sbyte`, е `SByte.MinValue` = -128 (-2^7), а максималната е `SByte.MaxValue` = 127 (2^7-1). Стойността по подразбиране е числото 0.

Типът `byte` е 8-битов **беззнаков** (**unsigned**) целочислен тип. Той също има 256 различни целочислени стойности (2^8), но те могат да бъдат само неотрицателни. Стойността по подразбиране на типа `byte` е числото 0. Минималната му стойност е `Byte.MinValue` = 0, а максималната е `Byte.MaxValue` = 255 (2^8-1).

Целочисленият тип `short` е 16-битов тип със знак. Минималната стойност, която може да заема, е `Int16.MinValue` = -32768 (-2^{15}), а максималната – `Int16.MaxValue` = 32767 (2^{15}-1). Стойността по подразбиране е числото 0.

Типът `ushort` е 16-битов беззнаков тип. Минималната стойност, която може да заема, е `UInt16.MinValue` = 0, а максималната – `UInt16.MaxValue` = 65535 (2^{16}-1). Стойността по подразбиране е числото 0.

Следващият целочислен тип, който ще разгледаме, е `int`. Той е 32- битов знаков тип. Както виждаме, с нарастването на битовете нарастват и възможните стойности, които даден тип може да заема. Стойността по подразбиране е числото 0. Минималната стойност, която може да заема, е `Int32.MinValue` = -2 147 483 648 (-2^{31}), а максималната е `Int32.MaxValue` = 2 147 483 647 (2^{31}-1).

Типът `int` е най-често използваният тип в програмирането. Обикновено програмистите използват `int`, когато работят с цели числа, защото този тип е естествен за 32-битовите микропроцесори и е достатъчно "голям" за повечето изчисления, които се извършват в ежедневието.

Типът `uint` е 32-битов беззнаков тип. Стойността по подразбиране е числото `0u` или `0U` (двата записа са еквивалентни). Символът '`u`' указва, че числото е от тип `uint` (иначе се подразбира `int`). Минималната стойност, която може да заема, е `UInt32.MinValue` = 0, а максималната му стойност е `UInt32.MaxValue` = 4 294 967 295 (2^{32}-1).

Типът `long` е 64-битов знаков тип със стойност по подразбиране `0l` или `0L` (двете са еквивалентни, но за предпочитане е да използвате '`L`', тъй като '`l`' лесно се бърка с цифрата единица '`1`'). Символът '`L`' указва, че числото е от тип `long` (иначе се подразбира `int`). Минималната стойност, която може да заема типът `long` е `Int64.MinValue` = -9 223 372 036 854 775 808 (-2^{63}), а максималната му стойност е `Int64.MaxValue` = 9 223 372 036 854 775 807 (2^{63}-1).

Най-големият целочислен тип е типът `ulong`. Той е 64-битов беззнаков тип със стойност по подразбиране е числото `0u` или `0U` (двата записа са еквивалентни). Символът '`u`' указва, че числото е от тип `ulong` (иначе се подразбира `long`). Минималната стойност, която може да бъде записана в типа `ulong` е `UInt64.MinValue` = 0, а максималната – `UInt64.MaxValue` = 18 446 744 073 709 551 615 (2^{64}-1).

Целочислени типове – пример

Нека разгледаме един пример, в който декларираме няколко променливи от познатите ни целочислени типове, инициализираме ги и отпечатваме стойностите им на конзолата:

```
// Declare some variables
byte centuries = 20;
ushort years = 2000;
uint days = 730480;
ulong hours = 17531520;
// Print the result on the console
Console.WriteLine(centuries + " centuries are " + years +
  " years, or " + days + " days, or " + hours + " hours.");

// Console output:
// 20 centuries are 2000 years, or 730480 days, or 17531520
// hours.

ulong maxIntValue = UInt64.MaxValue;
Console.WriteLine(maxIntValue); // 18446744073709551615
```

Какво представлява деклариране и инициализация на променлива, можем да прочетем в детайли по-долу в секциите "Деклариране на променливи" и "Инициализация на променливи", но това става ясно и от примерите.

В разгледания по-горе пример демонстрираме използването на целочислените типове. За малки числа използваме типа `byte`, а за много големи – `ulong`. Използваме беззнакови типове, тъй като всички използвани стойности са положителни числа.

Реални типове с плаваща запетая

Реалните типове в C# представляват реалните числа, които познаваме от математиката. Те се представят чрез **плаваща запетая** (**floating-point**) според стандарта IEEE 754 и биват `float` и `double`. Нека разгледаме тези два типа данни в детайли, за да разберем по какво си приличат и по какво се различават.

Реален тип float

Първият тип, който ще разгледаме, е 32-битовият реален тип с плаваща запетая `float`. Той се нарича още **реален тип с единична точност** (single precision real number). Стойността му по подразбиране е `0.0f` или `0.0F` (двете са еквивалентни). Символът '`f`' накрая указва изрично, че числото е от тип `float` (защото по подразбиране всички реални числа се приемат за `double`). Повече за този специален суфикс можем да прочетем в секцията "Реални литерали". Разглежданият тип има точност до 7 десетични знака (останалите се губят). Например числото `0.123456789` ако бъде записано в типа `float` ще бъде закръглено до `0.1234568`.

Диапазонът на стойностите, които могат да бъдат записани в типа **float** (със закръгляне до точност 7 значещи десетични цифри) е от **$\pm 1.5 \times 10^{-45}$** до **$\pm 3.4 \times 10^{38}$**.

Най-малката реална стойност на типа **float** е **Single.MinValue = -3.40282e+038f**, а най-голямата е **Single.MaxValue = 3.40282e+038f**. Най-близкото до 0 положително число от тип **float** е **Single.Epsilon = 4.94066e-324**.

Специални стойности на реалните типове

Реалните типове данни имат и няколко специални стойности, които не са реални числа, а представляват математически абстракции:

- Минус безкрайност -∞ (**Single.NegativeInfinity**). Получава се например като разделим **-1.0f** на **0.0f**.
- Плюс безкрайност +∞ (**Single.PositiveInfinity**). Получава се например като разделим **1.0f** на **0.0f**.
- Неопределеност (**Single.NaN**) – означава, че е извършена невалидна операция върху реални числа. Получава се например като разделим **0.0f** на **0.0f**, както и при кореноване на отрицателно число.

Реален тип double

Вторият реален тип с плаваща запетая в езика C# е типът **double**. Той се нарича още **реално число с двойна точност** (double precision real number) и представлява 64-битов тип със стойност по подразбиране **0.0d** или **0.0D** (символът '**d**' не е задължителен, тъй като по подразбиране всички реални числа в C# са от тип **double**). Разглежданият тип има точност от 15 до 16 десетични цифри. Диапазонът на стойностите, които могат да бъдат записани в **double** (със закръгляне до точност 15-16 значещи десетични цифри) е от **$\pm 5.0 \times 10^{-324}$** до **$\pm 1.7 \times 10^{308}$**.

Най-малката реална стойност на типа **double** е константата **Double.MinValue = -1.79769e+308**, а най-голямата – **Double.MaxValue = 1.79769e+308**. Най-близкото до 0 положително число от тип **double** е **Double.Epsilon = 4.94066e-324**. Както и при типа **float**, променливите от тип **double** могат да получават специалните стойности **Double.PositiveInfinity**, **Double.NegativeInfinity** и **Double.NaN**.

Реални типове – пример

Ето един пример, в който декларираме променливи от тип реално число, присвояваме им стойности и ги отпечатваме:

```
float floatPI = 3.14f;
Console.WriteLine(floatPI); // 3.14
double doublePI = 3.14;
```

```
Console.WriteLine(doublePI); // 3.14
double nan = Double.NaN;
Console.WriteLine(nan); // NaN
double infinity = Double.PositiveInfinity;
Console.WriteLine(infinity); // Infinity
```

Точност на реалните типове

Реалните числа в математиката в даден диапазон са неизброимо много (за разлика от целите числа), тъй като между всеки две реални числа **a** и **b** съществуват безброй много други реални числа **c**, за които **a < c < b**. Това налага необходимостта реалните числа да се съхраняват в паметта на компютъра с определена точност.

Тъй като математиката и най-вече физиката работят с изключително големи числа (положителни и отрицателни) и изключително малки числа (много близки до нула), е необходимо реалните типове в изчислителната техника да могат да ги съхраняват и обработват по подходящ начин. Например според физиката масата на електрона е приблизително $9.109389*10^{-31}$ килограма, а в един мол вещество има около $6.02*10^{23}$ атома. И двете посочени величини могат да бъдат записани безпроблемно в типовете **float** и **double**. Поради това удобство в съвременната изчислителна техника често се използва представянето с плаваща запетая – за да се даде възможност за работа с максимален брой значещи цифри при много големи числа (например положителни и отрицателни числа със стотици цифри) и при числа много близки до нулата (например положителни и отрицателни числа със стотици нули след десетичната запетая преди първата значеща цифра).

Точност на реални типове – пример

Разгледаните реални типове в C# **float** и **double** се различават освен с порядъка на възможните стойности, които могат да заемат, и по точност (броят десетични цифри, които запазват). Първият тип има точност 7 знака, вторият – 15-16 знака.

Нека разгледаме един пример, в който декларираме няколко променливи от познатите ни реални типове, инициализираме ги и отпечатваме стойностите им на конзолата. Целта на примера е да онагледим разликата в точността им:

```
// Declare some variables
float floatPI = 3.141592653589793238f;
double doublePI = 3.141592653589793238;

// Print the results on the console
Console.WriteLine("Float PI is: " + floatPI);
Console.WriteLine("Double PI is: " + doublePI);
```

```
// Console output:
// Float PI is: 3.141593
// Double PI is: 3.14159265358979
```

Виждаме, че числото **π**, което декларирахме от тип `float`, е закръглено на 7-мия знак, а при тип `double` – на 15-тия знак. Изводът, който можем да си направим е, че реалният тип `double` запазва доста по-голяма точност от `float` и ако ни е необходима голяма точност след десетичния знак, ще използваме него.

За представянето на реалните типове

Реалните числа с плаваща запетая в C# се състоят от три компонента (съгласно стандарта IEEE 754): **знак** (1 или -1), **мантиса** и **порядък** (**експонента**), като стойността им се изчислява по сложна формула. По-подробна информация за представянето на реалните числа сме предвидили в темата "Бройни системи", където разглеждаме в дълбочина представянето на числата и другите типове данни в изчислителната техника.

Грешки при пресмятания с реални типове

При пресмятания с реални типове данни с плаваща запетая е възможно да наблюдаваме странно поведение, тъй като при представянето на дадено реално число много често се губи точност. Причината за това е невъзможността някои реални числа да се представят точно сума от отрицателни степени на числото 2. Примери за числа, които нямат точно представяне в типовете `float` и `double`, са например 0.1, 1/3, 2/7 и други. Следва примерен C# код, който демонстрира грешките при пресмятания с числа с плаваща запетая в C#:

```
float f = 0.1f;
Console.WriteLine(f); // 0.1 (correct due to rounding)
double d = 0.1f;
Console.WriteLine(d); // 0.100000001490116 (incorrect)

float ff = 1.0f / 3;
Console.WriteLine(ff); // 0.3333333 (correct due to rounding)
double dd = ff;
Console.WriteLine(dd); // 0.333333343267441 (incorrect)
```

Причината за неочаквания резултат в първия пример е фактът, че числото 0.1 (т.е. 1/10) няма точно представяне във формата за реални числа с плаваща запетая IEEE 754 и се записва в него с приближение. При непосредствено отпечатване резултатът изглежда коректен заради закръглянето, което се извършва скрито при преобразуването на числото към стринг. При преминаване от `float` към `double` грешката, получена заради приближеното представяне на числото в IEEE 754 формат става вече явна и не може да бъде компенсирана от скритото закръгляване при отпечатването и съответно след осмата значеща цифра се появяват грешки.

При втория случай числото 1/3 няма точно представяне и се закръглява до число, много близко до 0.3333333. Кое е това число се вижда отчетливо, когато то се запише в типа **double**, който запазва много повече значещи цифри.

И двата примера показват, че аритметиката с числа с плаваща запетая може да прави грешки и по тази причина не е подходяща за прецизни финансови пресмятания. За щастие C# поддържа аритметика с десетична точност, при която числа като 0.1 се представят в паметта без закръгляне.

Не всички реални числа имат точно представяне в типовете `float` и `double`. Например числото 0.1 се представя закръглено в типа `float` като 0.099999994.

Реални типове с десетична точност

В C# се поддържа т. нар. **десетична аритметика с плаваща запетая (decimal floating-point arithmetic)**, при която числата се представят в десетична, а не в двоична бройна система и така не се губи точност при записване на десетично число в съответния тип с плаваща запетая.

Типът данни за реални числа с десетична точност в C# е 128-битовият тип **decimal**. Той има точност от 28 до 29 десетични знака. Минималната му стойност е **-7.9×10^{28}**, а максималната е **$+7.9\times10^{28}$**. Стойността му по подразбиране е **0.0м** или **0.0M**. Символът '**m**' накрая указва изрично, че числото е от тип **decimal** (защото по подразбиране всички реални числа са от тип **double**). Най-близките до 0 числа, които могат да бъдат записани в **decimal** са **$\pm1.0\times10^{-28}$**. Видно е, че **decimal** не може да съхранява много големи положителни и отрицателни числа (например със стотици цифри), нито стойности много близки до 0. За сметка на това този тип почти не прави грешки при финансови пресмятания, защото представя числата като сума от степени на числото 10, при което загубите от закръгляния са много по-малки, отколкото когато се използва двоично представяне. Реалните числа от тип **decimal** са изключително удобни за пресмятания с пари – изчисляване на приходи, задължения, данъци, лихви и т.н.

Следва пример, в който декларираме променлива от тип **decimal** и й присвояваме стойност:

```csharp
decimal decimalPI = 3.14159265358979323846m;
Console.WriteLine(decimalPI); // 3.14159265358979323846
```

Числото **decimalPI**, което декларирахме от тип **decimal**, не е закръглено дори и с един знак, тъй като го зададохме с точност 21 знака, което се побира в типа **decimal** без закръгляне.

Много голямата точност и липсата на аномалии при пресмятанията (каквито има при `float` и `double`) прави типа `decimal` много подходящ за финансови изчисления, където точността е критична.

Въпреки по-малкия си обхват, типът `decimal` запазва точност за всички десетични числа, които може да побере! Това го прави много подходящ за прецизни сметки, най-често финансови изчисления.

Основната разлика между реалните числа с плаваща запетая реалните числа с десетична точност е в точността на пресмятанията и в степента, до която те закръглят съхраняваните стойности. Типът `double` позволява работа с много големи стойности и стойности много близки до нулата, но за сметка на точността и неприятни грешки от закръгляне. Типът `decimal` има по-малък обхват, но гарантира голяма точност при пресмятанията и липсва на аномалии с десетичните числа.

Ако извършвате пресмятания с пари използвайте типа `decimal`, а не `float` или `double`. В противен случай може да се натъкнете на неприятни аномалии при пресмятанията и грешки в изчисленията!

Тъй като всички изчисления с данни от тип `decimal` се извършват чисто софтуерно, а не директно на ниско ниво в микропроцесора, изчисленията с този тип са от няколко десетки до стотици пъти по-бавни, отколкото същите изчисления с `double`, така че ползвайте този тип внимателно.

Булев тип

Булевият тип се декларира с ключовата дума `bool`. Той има две стойности, които може да приема – `true` и `false`. Стойността по подразбиране е `false`. Използва се най-често за съхраняване на резултата от изчисляването на логически изрази.

Булев тип – пример

Нека разгледаме един пример, в който декларираме няколко променливи от вече познатите ни типове, инициализираме ги, извършваме сравнения върху тях и отпечатваме резултатите на конзолата:

```
// Declare some variables
int a = 1;
int b = 2;
// Which one is greater?
bool greaterAB = (a > b);
// Is 'a' equal to 1?
bool equalA1 = (a == 1);
// Print the results on the console
```

```
if (greaterAB)
{
  Console.WriteLine("A > B");
}
else
{
  Console.WriteLine("A <= B");
}
Console.WriteLine("greaterAB = " + greaterAB);
Console.WriteLine("equalA1 = " + equalA1);

// Console output:
// A <= B
// greaterAB = False
// equalA1 = True
```

В примера декларираме две променливи от тип **`int`**, сравняваме ги и присвояваме резултата на променливата от булев тип **`greaterAB`**. Аналогично извършваме и за променливата **`equalA1`**. Ако променливата **`greaterAB`** е **`true`**, на конзолата се отпечатва **A > B**, в противен случай се отпечатва **A <= B**.

Символен тип

Символният тип представя единичен символ (16-битов номер на знак от Unicode таблицата). Той се декларира с ключовата дума **`char`** в езика C#. **Unicode таблицата** е технологичен стандарт, който съпоставя цяло число или поредица от няколко цели числа на всеки знак от човешките писмености по света (всички езици и техните азбуки). Повече за Unicode таблицата можем да прочетем в темата "Символни низове". Минималната стойност, която може да заема типът **`char`**, е 0, а максималната – 65535. Стойностите от тип **`char`** представляват букви или други символи и се ограждат в апострофи.

Символен тип – пример

Нека разгледаме един пример, в който декларираме една променлива от тип **`char`**, инициализираме я със стойност **`'a'`**, след това с **`'b'`** и **`'A'`** и отпечатваме Unicode стойностите на тези букви на конзолата:

```
// Declare a variable
char symbol = 'a';
// Print the results on the console
Console.WriteLine(
  "The code of '" + symbol + "' is: " + (int)symbol);
symbol = 'b';
Console.WriteLine(
  "The code of '" + symbol + "' is: " + (int)symbol);
symbol = 'A';
```

```
Console.WriteLine(
  "The code of '" + symbol + "' is: " + (int)symbol);

// Console output:
// The code of 'a' is: 97
// The code of 'b' is: 98
// The code of 'A' is: 65
```

Символни низове (стрингове)

Символните низове представляват поредица от символи. Декларират се с ключовата дума **`string`** в C#. Стойността им по подразбиране е **`null`**. Стринговете се ограждат в двойни кавички. Върху тях могат да се извършват различни текстообработващи операции: конкатениране (долепване един до друг), разделяне по даден разделител, търсене, знакозаместване и други. Повече информация за текстообработката можем да прочетем в темата "Символни низове", в която детайлно е обяснено какво е символен низ, за какво служи и как да го използваме.

Символни низове – пример

Нека разгледаме един пример, в който декларираме няколко променливи от тип символен низ, инициализираме ги и отпечатваме стойностите им на конзолата:

```
// Declare some variables
string firstName = "Ivan";
string lastName = "Ivanov";
string fullName = firstName + " " + lastName;
// Print the results on the console
Console.WriteLine("Hello, " + firstName + "!");
Console.WriteLine("Your full name is " + fullName + ".");

// Console output:
// Hello, Ivan!
// Your full name is Ivan Ivanov.
```

Обектен тип

Обектният тип е специален тип, който се явява родител на всички други типове в .NET Framework. Декларира се с ключовата дума **`object`** и може да приема стойности от всеки друг тип. Той представлява референтен тип, т.е. указател (адрес) към област от паметта, която съхранява неговата стойност.

Използване на обекти – пример

Нека разгледаме един пример, в който декларираме няколко променливи от обектен тип, инициализираме ги и отпечатваме стойностите им на конзолата:

```
// Declare some variables
object container1 = 5;
object container2 = "Five";

// Print the results on the console
Console.WriteLine("The value of container1 is: " + container1);
Console.WriteLine("The value of container2 is: " + container2);

// Console output:
// The value of container is: 5
// The value of container2 is: Five.
```

Както се вижда от примера, в променлива от тип **`object`** можем да запишем стойност от всеки друг тип. Това прави обектния тип универсален контейнер за данни.

Нулеви типове (Nullable Types)

Нулевите типове (**nullable types**) представляват специфични обвивки (**wrappers**) около стойностните типове (като **`int`**, **`double`** и **`bool`**), които позволяват в тях да бъде записвана **`null`** стойност. Това дава възможност в типове, които по принцип не допускат липса на стойност (т.е. стойност **`null`**), все пак да могат да бъдат използвани като референтни типове и да приемат както нормални стойности, така и специалната стойност **`null`**.

Обвиването на даден тип като нулев става по два начина:

```
Nullable<int> i1 = null;
int? i2 = i1;
```

Двете декларации са еквивалентни. По-лесният начин е да се добави въпросителен знак (**`?`**) след типа, например **`int?`**, а по-трудният е да се използва **`Nullable<…>`** синтаксиса.

Нулевите типове са референтни типове, т.е. представляват референция към обект в динамичната памет, който съдържа стойността им. Те могат да имат или нямат стойност и могат да бъдат използвани както нормалните примитивни типове, но с някои особености, които ще илюстрираме в следващия пример:

```
int i = 5;
int? ni = i;
Console.WriteLine(ni); // 5

// i = ni; // this will fail to compile
Console.WriteLine(ni.HasValue); // True
i = ni.Value;
Console.WriteLine(i); // 5
```

```
ni = null;
Console.WriteLine(ni.HasValue); // False
//i = ni.Value; // System.InvalidOperationException
i = ni.GetValueOrDefault();
Console.WriteLine(i); // 0
```

От примера е видно, че на променлива от нулев тип (**`int?`**) може да се присвои директно стойност от ненулев тип (**`int`**), но обратното не е директно възможно. За целта може да се използва свойството на нулевите типове **`Value`**, което връща стойността записана в нулевия тип или предизвиква грешка (**`InvalidOperationException`**) по време на изпълнение на програмата, ако стойност липсва. За да проверим дали променлива от нулев тип има стойност, можем да използваме булевото свойство **`HasValue`**. Ако искаме да вземем стойността променлива от нулев тип или стойността за типа по подразбиране (най-често `0`) в случай на **`null`**, можем да използваме метода **`GetValueOrDefault()`**.

Нулевите типове се използват за съхраняване на информация, която не е задължителна. Например, ако искаме да запазим данните за един студент, като името и фамилията му са задължителни, а възрастта му не е задължителна, можем да използваме **`int?`** за възрастта:

```
string firstName = "Svetlin";
string lastName = "Nakov";
int? age = null;
```

Променливи

След като разгледахме основните типове данни в C#, нека видим как и за какво можем да ги използваме. За да работим с данни, трябва да използваме променливи. Вече се сблъскахме с променливите в примерите, но сега нека ги разгледаме по-подробно.

Променливата е контейнер на информация, който може да променя стойността си. Тя осигурява възможност за:

- запазване на информация;
- извличане на запазената информация;
- модифициране на запазената информация.

Програмирането на C# е свързано с постоянно използване на променливи, в които се съхраняват и обработват данни.

Характеристики на променливите

Променливите се характеризират с:

- име (идентификатор), например **`age`**;

- тип (на запазената в тях информация), например **`int`**;
- стойност (запазената информация), например `25`.

Една променлива представлява именувана област от паметта, в която е записана стойност от даден тип, достъпна в програмата по своето име. Променливите могат да се пазят непосредствено в работната памет на програмата (в стека) или в динамичната памет, която се съхраняват по-големи обекти (например символни низове и масиви). Примитивните типове данни (числа, **`char`**, **`bool`**) се наричат стойностни типове, защото пазят непосредствено своята стойност в стека на програмата. Референтните типове данни (например стрингове, обекти и масиви) пазят като стойност адрес от динамичната памет, където е записана стойността им. Те могат да се заделят и освобождават динамично, т.е. размерът им не е предварително фиксиран, както при стойностните типове. Повече информация за стойностите и референтните типове данни сме предвидили в секцията "Стойностни и референтни типове".

Именуване на променлива – правила

Когато искаме компилаторът да задели област в паметта за някаква информация, използвана в програмата ни, трябва да и зададем име. То служи като идентификатор и позволява да се реферира нужната ни област от паметта.

Името на променливите може да бъде всякакво по наш избор, но трябва да следва определени правила:

- Имената на променливите се образуват от буквите **`a-z`**, **`A-Z`**, цифрите `0-9`, както и символа '_'.
- Имената на променливите не може да започват с цифра.
- Имената на променливите не могат да съвпадат със **служебна дума** (**keyword**) от езика C#.

В следващата таблица са дадени всички служебни думи в C#. Някои от тях вече са ни известни, а с други ще се запознаем в следващите глави от книгата:

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	in (generic)	int
interface	internal	is	lock	long
namespace	new	null	object	operator

out	**out (generic)**	**override**	**params**	**private**
protected	**public**	**readonly**	**ref**	**return**
sbyte	**sealed**	**short**	**sizeof**	**stackalloc**
static	**string**	**struct**	**switch**	**this**
throw	**true**	**try**	**typeof**	**uint**
ulong	**unchecked**	**unsafe**	**ushort**	**using**
virtual	**void**	**volatile**	**while**	

Именуване на променливи – примери

Позволени имена:

- **`name`**
- **`first_Name`**
- **`_name1`**

Непозволени имена (ще доведат до грешка при компилация):

- `1` (цифра)
- **`if`** (служебна дума)
- **`1name`** (започва с цифра)

Именуване на променливи – препоръки

Ще дадем някои препоръки за именуване, тъй като не всички позволени от компилатора имена са подходящи за нашите променливи.

- Имената трябва да са описателни и да обясняват за какво служи дадената променлива. Например за име на човек подходящо име е **`personName`**, а неподходящо име е **`a37`**.
- Трябва да се използват само латински букви. Въпреки, че кирилицата е позволена от компилатора, не е добра практика тя да бъде използвана в имената на променливите и останалите идентификатори от програмата.
- В C# е прието променливите да започват винаги с малка буква и да съдържат малки букви, като всяка следваща дума в тях започва с главна буква. Например правилно име е **`firstName`**, а не **`firstname`** или **`first_name`**. Използването на символа **`_`** в имената на променливите се счита за лош стил на именуване.
- Името на променливите трябва да не е нито много дълго, нито много късо – просто трябва да е ясно за какво служи променливата в контекста, в който се използва.
- Трябва да се внимава за главни и малки букви, тъй като C# прави разлика между тях. Например **`age`** и **`Age`** са различни променливи.

Ето няколко примера за добре именувани променливи:

- **`firstName`**
- **`age`**
- **`startIndex`**
- **`lastNegativeNumberIndex`**

Ето няколко примера за лошо именувани променливи (макар и имената да са коректни от гледана точка на компилатора на C#):

- **`_firstName`** (започва с _)
- **`last_name`** (съдържа _)
- **`AGE`** (изписана е с главни букви)
- **`Start_Index`** (започва с главна буква и съдържа _)
- **`lastNegativeNumber_Index`** (съдържа _)

Променливите трябва да имат име, което обяснява накратко за какво служат. Когато една променлива е именувана с неподходящо име, това силно затруднява четенето на програмата и нейната следваща промяна (след време, когато сме забравили как работи тя). Повече за правилното именуване на променливите ще научите в главата "Качествен програмен код".

Стремете се винаги да именувате променливите с кратки, но достатъчно ясни имена. Следвайте правилото, че от името на променливата трябва да става ясно за какво се използва, т.е. името трябва да отговаря на въпроса "каква стойност съхранява тази променлива". Ако това не е изпълнено, потърсете по-добро име.

Деклариране на променливи

Когато декларираме променлива, извършваме следните действия:

- задаваме нейния тип (например **`int`**);
- задаваме нейното име (идентификатор, например **`age`**);
- можем да зададем начална стойност (например 25), но това не е задължително.

Синтаксисът за деклариране на променливи в C# е следният:

```
<тип данни> <идентификатор> [= <инициализация>]
```

Ето един пример за деклариране на променливи:

```
string name;
int age;
```

Присвояване на стойност

Присвояването на стойност на променлива представлява задаване на стойност, която да бъде записана в нея. Тази операция се извършва чрез оператора за присвояване '='. От лявата страна на оператора се изписва име на променлива, а от дясната страна – новата й стойност.

Ето един пример за присвояване на стойност на променливи:

```
name = "Svetlin Nakov";
age = 25;
```

Инициализация на променливи

Терминът **инициализация** в програмирането означава задаване на начална стойност. Задавайки стойност на променливите в момента на тяхното деклариране, ние всъщност ги инициализираме.

Всеки тип данни в C# има стойност по подразбиране (инициализация по подразбиране), която се използва, когато за дадена променлива не бъде изрично зададена стойност. Можем да си припомним стойностите по подразбиране за типовете, с които се запознахме, от следващата таблица:

Тип данни	Стойност по подразбиране	Тип данни	Стойност по подразбиране
sbyte	0	**float**	0.0f
byte	0	**double**	0.0d
short	0	**decimal**	0.0m
ushort	0	**bool**	false
int	0	**char**	'\u0000'
uint	0u	**string**	null
long	0L	**object**	null
ulong	0u		

Нека обобщим как декларираме променливи, как ги инициализираме и как им присвояваме стойности в следващия пример:

```
// Declare and initialize some variables
byte centuries = 20;
ushort years = 2000;
decimal decimalPI = 3.141592653589793238m;
bool isEmpty = true;
char symbol = 'a';
string firstName = "Ivan";

symbol = (char)5;
```

```
char secondSymbol;

// Here we use an already initialized variable and reassign it
secondSymbol = symbol;
```

Стойностни и референтни типове

Типовете данни в C# са 2 вида: **стойностни** и **референтни**.

Стойностните типове (value types) се съхраняват в стека за изпълнение на програмата и съдържат директно стойността си. Стойностни са примитивните числови типове, символният тип и булевият тип: **`sbyte`**, **`byte`**, **`short`**, **`ushort`**, **`int`**, **`long`**, **`ulong`**, **`float`**, **`double`**, **`decimal`**, **`char`**, **`bool`**. Те се освобождават при излизане от обхват, т.е. когато блокът с код, в който са дефинирани, завърши изпълнението си. Например една променлива, декларирана в метода **`Main()`** на програмата се пази в стека докато програмата завърши изпълнението на този метод, т.е. докато не завърши.

Референтните типове (reference types) съдържат в стека за изпълнение на програмата референция (адрес) към **динамичната памет** (**heap**), където се съхранява тяхната стойност. Референцията представлява **указател** (адрес на клетка от паметта), сочещ реалното местоположение на стойността в динамичната памет. Пример за стойност на адрес в стека за изпълнение е **`0x00AD4934`**. Референцията има тип и може да съдържа като стойност само обекти от своя тип, т.е. тя представлява строго типизиран указател. Всички референтни типове могат да получават стойност **`null`**. Това е специална служебна стойност, която означава, че липсва стойност.

Референтните типове заделят динамична памет при създаването си и се освобождават по някое време от системата за **почистване на паметта (garbage collector)**, когато тя установи, че вече не се използват от програмата. Не е известно точно в кой момент дадена референтна променлива ще бъде освободена от garbage collector, тъй като това зависи от натоварването на паметта и от други фактори. Тъй като заделянето и освобождаването на памет е бавна операция, може да се каже, че референтните типове са по-бавни от стойностните.

Тъй като референтните типове данни се заделят и освобождават динамично по време на изпълнение на програмата, техният размер може да не е предварително известен. Например в променлива от тип **`string`** могат да бъдат записвани текстови данни с различна дължина. Реално текстовата стойност на типа **`string`** се записва в динамичната памет и може да заема различен обем (брой байтове), а в променливата от тип **`string`** се записва адресът неговият адрес.

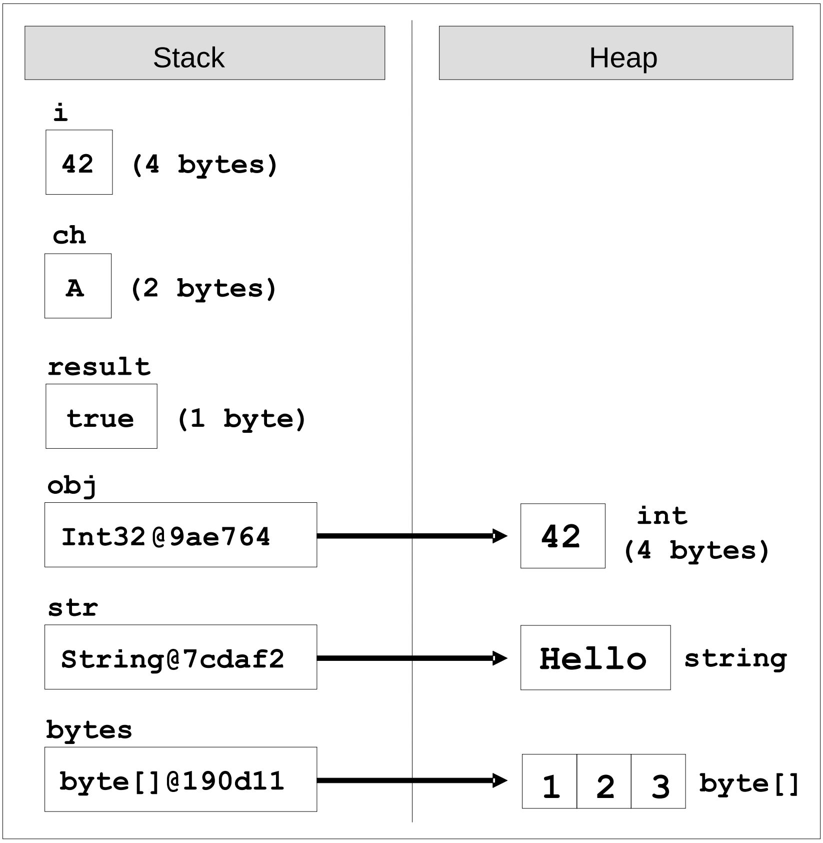
Референтни типове са всички класове, масивите и интерфейсите, например типовете: **`object`**, **`string`**, **`byte[]`**. С класовете, обектите, символните низове, масивите и интерфейсите ще се запознаем в следващите глави на книгата. Засега е достатъчно да знаете, че всички типове, които не са стойностни, са референтни и се разполагат в динамичната памет.

Стойностни и референтни типове и паметта

Нека илюстрираме с един пример как се представят в паметта стойностните и референтните типове. Нека е изпълнен следния програмен код:

```
int i = 42;
char ch = 'A';
bool result = true;
object obj = 42;
string str = "Hello";
byte[] bytes = { 1, 2, 3 };
```

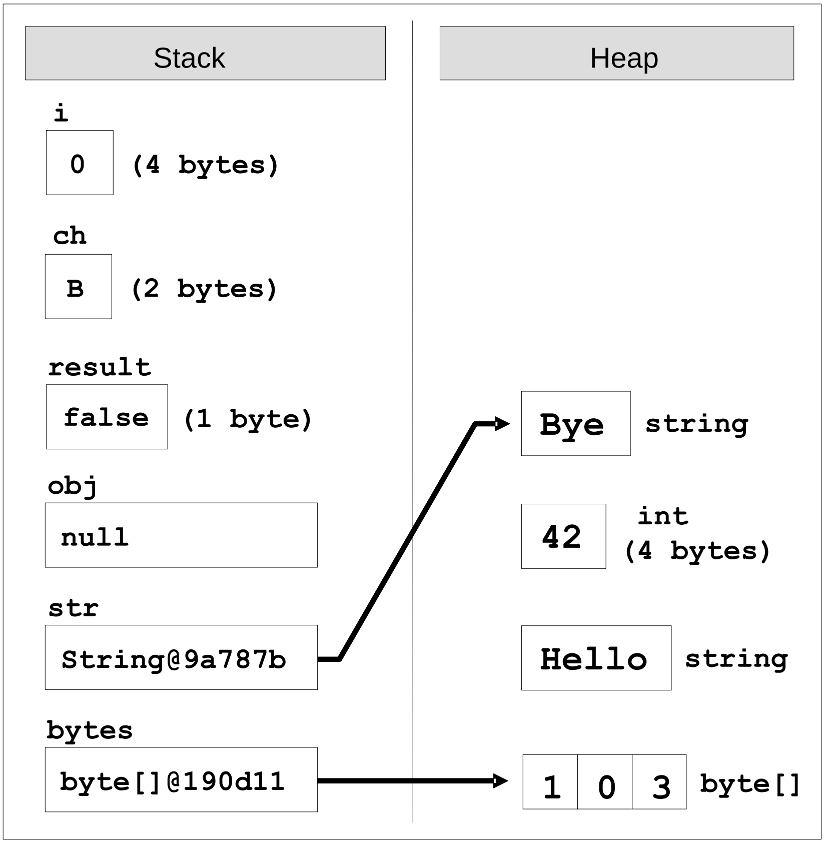
В този момент променливите са разположени в паметта по следния начин:


Stack
Heap
i
42
(4 bytes)
ch
A
(2 bytes)
result
true
(1 byte)
obj
Int32@9ae764
42
int
(4 bytes)
str
String@7cdaf2
Hello
string
bytes
byte[]@190d11
1
2
3
byte[]

Ако сега изпълним следния код, който променя стойностите на променливите, ще видим какво се случва с паметта при промяна на стойностни и референтни типове:

```
i = 0;
ch = 'B';
result = false;
obj = null;
str = "Bye";
bytes[1] = 0;
```

След тези промени променливите и техните стойности са разположени в паметта по следния начин:


Stack
Heap
i
0
(4 bytes)
ch
B
(2 bytes)
result
false
(1 byte)
obj
null
str
String@9a787b
bytes
byte[]@190d11
Bye
string
42
int
(4 bytes)
Hello
string
1
0
3
byte[]

Както можете да забележите от фигурата, при промяна на стойностен тип (**i=0**) се променя директно стойността му в стека. При промяна на референтен тип нещата са по-различни: променя се директно стойността му в динамичната памет (**bytes[1]=0**). Променливата, която държи референцията, остава непроменена (**0x00AD4934**). При записване на стойност **null** в референтен тип съответната референция се разкача от стойността си и променливата остава без стойност (**obj=null**).

При присвояване на нова стойност на обект (променлива от референтен тип), новият обект се заделя в динамичната стойност, а старият обект остава свободен (нерефериран). Референцията се пренасочва към новия обект (**str="Bye"**), а старите обекти (**"Hello"**), понеже не се използват, ще бъдат почистени по някое време от системата за почистване на паметта (garbage collector).

Литерали

Примитивните типове, с които се запознахме вече, са специални типове данни, вградени в езика C#. Техните стойности, зададени в сорс кода на програмата, се наричат **литерали**. С един пример ще ни стане по-ясно:

```
bool result = true;
char capitalC = 'C';
byte b = 100;
short s = 20000;
int i = 300000;
```

В примера литерали са `true`, `'C'`, `100`, `20000` и `300000`. Те представляват стойности на променливи, зададени непосредствено в сорс кода на програмата.

Видове литерали

В езика C# съществуват няколко вида литерали:

- булеви
- целочислени
- реални
- символни
- низови
- обектният литерал `null`

Булеви литерали

Булевите литерали са:

- `true`
- `false`

Когато присвояваме стойност на променлива от тип `bool`, можем да използваме единствено някоя от тези две стойности или израз от булев тип (който се изчислява до `true` или `false`).

Булеви литерали – пример

Ето пример за декларация на променлива от тип `bool` и присвояване на стойност, което представлява булевият литерал `true`:

```
bool result = true;
```

Целочислени литерали

Целочислените литерали представляват поредица от цифри, знак (**+**, **-**), окончания и представки. С помощта на представки можем да представим целите числа в сорс кода на програмата в десетичен или шестнадесетичен формат. Повече информация за различните бройни системи можем да получим в темата "Бройни системи". В целочислените литерали могат да участват и следните представки и окончания:

- "**0x**" и "**0X**" като представки означават шестнадесетична стойност, например **0xA8F1**;
- '**l**' и '**L**' като окончания означават данни от тип **long**, например **357L**.
- '**u**' и '**U**' като окончания означават данни от тип **uint** или **ulong**, например **112u**.

По подразбиране (ако не бъде използвана никакво окончание) целочислените литерали са от тип **int**.

Целочислени литерали – примери

Ето няколко примера за използване на целочислени литерали:

```
// The following variables are initialized with the same value
int numberInDec = 16;
int numberInHex = 0x10;

// This will cause an error, because the value 234L is not int
int longInt = 234L;
```

Реални литерали

Реалните литерали, представляват поредица от цифри, знак (**+**, **-**), окончания и символа за десетична запетая. Използваме ги за стойности от тип **float**, **double** и **decimal**. Реалните литерали могат да бъдат представени и в експоненциален формат. При тях се използват още следните означения:

- '**f**' и '**F**' като окончания означават данни от тип **float**;
- '**d**' и '**D**' като окончания означават данни от тип **double**;
- '**m**' и 'M' като окончания означават данни от тип **decimal**;
- '**e**' означава експонента, например "e-5" означава цялата част да се умножи по 10^{-5}.

По подразбиране (ако липсва окончание) реалните числа са от тип **double**.

Реални литерали – примери

Ето няколко примера за използване на реални литерали:

```
// The following is the correct way of assigning a value:
float realNumber = 12.5f;

// This is the same value in exponential format:
realNumber = 1.25e+1f;

// The following causes an error, because 12.5 is double
float realNumber = 12.5;
```

Символни литерали

Символните литерали представляват единичен символ, ограден в апострофи (единични кавички). Използваме ги за задаване на стойности от тип **`char`**. Стойността на символните литерали може да бъде:

- символ, примерно **`'A'`**;
- код на символ, примерно **`'\u0065'`**;
- escaping последователност;

Екранирани (Escaping) последователности

Понякога се налага да работим със символи, които не са изписани на клавиатурата или със символи, които имат специално значение, като например символът "нов ред". Те не могат да се изпишат директно в сорс кода на програмата и за да ги ползваме са ни необходими специални техники, които ще разгледаме сега.

Escaping последователностите са литерали, които представляват последователност от специални символи, които задават символ, който по някаква причина не може да се изпише директно в сорс кода. Такъв е например символът за нов ред. Те ни дават заобиколен начин (**escaping**) да напишем някакъв символ на екрана и затова се наричат още **екранирани последователности**.

Примери за символи, които не могат да се изпишат директно в сорс кода, има много: двойна кавичка, табулация, нов ред, наклонена черта и други. Ето някои от най-често използваните escaping последователности:

- **`\'`** – единична кавичка
- **`\"`** – двойна кавичка
- **`\\`** – лява наклонена черта
- **`\n`** – нов ред
- **`\t`** – отместване (табулация)

- `\uXXXX` – символ, зададен с Unicode номера си, примерно `\u03A7`.

Символът \ (лява наклонена черта) се нарича още **екраниращ символ** (**escaping character**), защото той позволява да се изпишат на екрана символи, които имат специално значение или действие и не могат да се изпишат директно в сорс кода.

Escaping последователности – примери

Ето няколко примера за символни литерали:

```
// An ordinary symbol
char symbol = 'a';
Console.WriteLine(symbol);

// Unicode symbol code in a hexadecimal format
symbol = '\u003A';
Console.WriteLine(symbol);

// Assigning the single quote symbol (escaped as \')
symbol = '\'';
Console.WriteLine(symbol);

// Assigning the backslash symbol(escaped as \\)
symbol = '\\';
Console.WriteLine(symbol);

// Console output:
// a
// :
// '
// \
```

Литерали за символен низ

Литералите за символен низ се използват за данни от тип `string`. Те представляват последователност от символи, заградена в двойни кавички.

За символните низове важат всички правила за escaping, които вече разгледахме за литералите от тип `char`.

Символните низове могат да се изписват предхождани от символа @, който задава **цитиран низ**. В цитираните низове правилата за escaping не важат, т.е. символът \ означава \ и не е екраниращ символ. В цитираните низове кавичката " може да се екранира с двойна "", а всички останали символи се възприемат буквално, дори новият ред. Цитираните низове се използват често пъти при задаване на имена на пътища във файловата система.

Литерали за символен низ – примери

Ето няколко примера за използване на литерали от тип символен низ:

```
string quotation = "\"Hello, Jude\", he said.";
Console.WriteLine(quotation);
string path = "C:\\Windows\\Notepad.exe";
Console.WriteLine(path);
string verbatim = @"The \ is not escaped as \\.
I am at a new line.";
Console.WriteLine(verbatim);

// Console output:
// "Hello, Jude", he said.
// C:\Windows\Notepad.exe
// The \ is not escaped as \\.
// I am at a new line.
```

Повече за символните низове ще намерим в темата "Символни низове".

Упражнения

1. Декларирайте няколко променливи, като изберете за всяка една най-подходящия от типовете **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long** и **ulong**, за да им присвоите следните стойности: 52130, -115, 4825932, 97, -10000, 20000; 224; 970700000; 112; -44; -1000000; 1990; 123456789123456789.
2. Кои от следните стойности може да се присвоят на променливи от тип **float**, **double** и **decimal**: 34.567839023; 12.345; 8923.1234857; 3456.091124875956542151256683467?
3. Напишете програма, която изчислява вярно променливи с плаваща запетая с точност до 0.000001.
4. Инициализирайте променлива от тип **int** със стойност 256 в шестнадесетичен формат (256 е 100 в бройна система с основа 16).
5. Декларирайте променлива от тип **char** и присвоете като стойност символа който има Unicode код 72 (използвайте калкулатора на Windows за да намерите шестнайсетичното представяне на 72).
6. Декларирайте променлива **isMale** от тип **bool** и присвоете стойност на последната в зависимост от вашия пол.
7. Декларирайте две променливи от тип **string** със стойности "Hello" и "World". Декларирайте променлива от тип **object**. Присвоете на тази променлива стойността, която се получава от конкатенацията на двете стрингови променливи (добавете интервал, ако е необходимо). Отпечатайте променливата от тип **object**.
8. Декларирайте две променливи от тип **string** и им присвоете стойности "Hello" и "World". Декларирайте променлива от тип **object** и и присвоете стойността на конкатенацията на двете променливи от тип **string** (не изпускайте интервала по средата). Декларирайте

трета променлива от тип **`string`** и я инициализирайте със стойността на променливата от тип **`object`** (трябва да използвате type casting).

9. Декларирайте две променливи от тип **`string`** и им присвоете стойност "The "use" of quotations causes difficulties." (без първите и последни кавички). В едната променлива използвайте quoted string, а в другата не го използвайте.
10. Напишете програма, която принтира фигура във формата на сърце със знака "o".
11. Напишете програма, която принтира на конзолата равнобедрен триъгълник, като страните му са очертани от символа звездичка "©".
12. Фирма, занимаваща се с маркетинг, иска да пази запис с данни на нейните служители. Всеки запис трябва да има следната характеристика – първо име, фамилия, възраст, пол (‘м’ или ‘ж’) и уникален номер на служителя (27560000 до 27569999). Декларирайте необходимите променливи, нужни за да се запази информацията за един служител, като използвате подходящи типове данни и описателни имена.
13. Декларирайте две променливи от тип **`int`**. Задайте им стойности съответно 5 и 10. Разменете стойностите им и ги отпечатайте.

Решения и упътвания

1. Погледнете размерността на числените типове.
2. Имайте предвид броя символи след десетичния знак. Направете справка в таблицата с размерите на типовете **`float`**, **`double`** и **`decimal`**.
3. Използвайте типа данни **`decimal`**.
4. Вижте секцията за целочислени литерали. За да преобразувате лесно числата в различна бройна система използвайте вградения в Windows калкулатор. За шестнайсетично представяне на литерал използвайте префикса **`0x`**.
5. Вижте секцията за целочислени литерали.
6. Вижте секцията за булеви променливи.
7. Вижте секциите за символни низове и за обектен тип данни.
8. Вижте секциите за символни низове и за обектен тип данни.
9. Погледнете частта за символни литерали. Необходимо е да използвате символа за escaping (наклонена черта "\").
10. Използвайте **`Console.Writeline(…)`** като използвате само символа ‘o’ и интервали.

11. Използвайте **`Console.Writeline(…)`** като използвате само знака © и интервали. Използвайте Windows Character Map, за да намерите Unicode кода на знака "©".

12. За имената използвайте тип **`string`**, за пола използвайте тип **`char`** (имаме само един символ м/ж), а за уникалния номер и възрастта използвайте подходящ целочислен тип.

13. Използвайте трета временна променлива за размяната на променливи. За целочислените променливи е възможно и друго решение, което не използва трета променлива. Например, ако имаме 2 променливи **`a`** и **`b`**:

```
int a = 2;
int b = 3;
a = a + b;
b = a - b;
a = a - b;
```

Глава 3. Оператори и изрази

В тази тема...

В настоящата тема ще се запознаем с операторите в C# и действията, които те извършват върху различните типове данни. Ще разясним приоритетите на операторите и ще разгледаме видовете оператори според броя на аргументите, които приемат и какви действия извършват. Във втората част на темата ще разгледаме преобразуването на типове, ще обясним кога и защо се налага да се извършва и как да работим с типове. В края на темата ще обърнем внимание на изразите и как да работим с тях. Най-накрая сме приготвили упражнения, за да затвърдим знанията си по материала от тази глава.

Оператори

Във всички езици за програмиране се използват оператори, чрез които се извършват някакви действия върху данните. Нека разгледаме операторите в C# и да видим за какво служат и как се използват.

Какво е оператор?

След като научихме как да декларираме и да задаваме стойности на променливи в предходната глава, ще разгледаме как да извършваме различни операции върху тях. За целта ще се запознаем с операторите.

Операторите позволят обработка на примитивни типове данни и обекти. Те приемат като вход един или няколко операнда и връщат като резултат някаква стойност. Операторите в C# представляват специални символи (като например "+", ".", "^" и други) и извършат специфични преобразувания над един, два или три операнда. Пример за оператори в C# са знаците за събиране, изваждане, умножение и делене в математиката (+, - , *, /) и операциите, които те извършват върху целите и реалните числа.

Операторите в C#

Операторите в C# могат да бъдат разделени в няколко различни категории:

- Аритметични – също както в математиката, служат за извършване на прости математически операции.
- Оператори за присвояване – позволяват присвояването на стойност на променливите.
- Оператори за сравнение – дават възможност за сравнение на два литерала и/или променливи.
- Логически оператори – оператори за работа с булеви типове данни и булеви изрази.
- Побитови оператори – използват се за извършване на операции върху двоичното представяне на числови данни.
- Оператори за преобразуване на типовете – позволяват преобразуването на данни от един тип в друг.

Категории оператори

Следва списък с операторите, разделени по категории:

Категория	**Оператори**
аритметични	`-, +, *, /, %, ++, --`

логически	**&&, ||, !, ^**
побитови	**&, |, ^, ~, <<, >>**
за сравнение	**==, !=, >, <, >=, <=**
за присвояване	**=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=**
съединяване на символни низове	**+**
за работа с типове	**(type), as, is, typeof, sizeof**
други	**., new, (), [], ?:, ??**

Оператори според броя аргументи

Операторите могат да се разделят на типове според броя на аргументите, които приемат:

Тип оператор	Брой на аргументите (операндите)
едноаргументни (unary)	приема един аргумент
двуаргументни (binary)	приема два аргумента
триаргументни (ternary)	приема три аргумента

Всички двуаргументни оператори в C# са ляво-асоциативни, т.е. изразите, в които участват се изчисляват от ляво на дясно, освен операторите за присвояване на стойности. Всички оператори за присвояване на стойности и условните оператори **?:** и **??** са дясно-асоциативни (изчисляват се от дясно на ляво). Едноаргументните оператори нямат асоциативност.

Някой оператори в C# извършват различни операции, когато се приложат върху различен тип данни. Пример за това е операторът **+**. Когато се използва върху числени типове данни (**int**, **long**, **float** и др.), операторът извършва операцията математическо събиране. Когато обаче използваме оператора върху символни низове, той слепва съдържанието на двете променливи / литерали и връща новополучения низ.

Оператори – пример

Ето един пример за използване на оператори:

```
int a = 7 + 9;
Console.WriteLine(a); // 16

string firstName = "Dilyan";
string lastName = "Dimitrov";

// Do not forget the interval between them
string fullName = firstName + " " + lastName;
```

```
Console.WriteLine(fullName); // Dilyan Dimitrov
```

Примерът показва как при използването на оператора **+** върху числа той връща числова стойност, а при използването му върху низове връща низ.

Приоритет на операторите в C#

Някои оператори имат **приоритет** над други. Например, както е в математиката, умножението има приоритет пред събирането. Операторите с по-висок приоритет се изчисляват преди тези с по-нисък. Операторът `()` служи за промяна на приоритета на операторите и се изчислява пръв, също както в математиката.

В таблицата са показани приоритетите на операторите в C#:

Приоритет	Оператори
най-висок	`++`, `--` (като постфикс), `new`, `(type)`, `typeof`, `sizeof`
	`++`, `--` (като префикс), `+`, `-` (едноаргументни), `!`, `~`
	`*`, `/`, `%`
	`+` (свързване на низове)
	`+`, `-`
	`<<`, `>>`
…	`<`, `>`, `<=`, `>=`, `is`, `as`
	`==`, `!=`
	`&`, `^`, `|`
	`&&`
	`||`
най-нисък	`?:`, `??`
	`=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`

Операторите, намиращи се по-нагоре в таблицата, имат по-висок приоритет от тези, намиращи се след тях, и съответно имат предимство при изчисляването на даден израз. За да променим приоритета на даден оператор може да използваме скоби.

Когато пишем по-сложни изрази или такива съдържащи повече оператори се препоръчва използването на скоби, за да се избегнат трудности при четене и разбиране на кода. Ето един пример:

```
// Ambiguous
x + y / 100
```

```
// Unambiguous, recommended
x + (y / 100)
```

Първата операция, която се изпълнява в примера, е делението, защото то има по-висок приоритет от оператора за събиране. Въпреки това използването на скоби е добра идея, защото кодът става по-лесен за четене и възможността да се допусне грешка намалява.

Аритметични оператори

Аритметичните оператори в C# **+**, **-**, ***** са същите като в математика. Те извършват съответно събиране, изваждане и умножение върху числови стойности и резултатът е отново целочислена стойност.

Операторът за деление **/** има различно действие върху цели и реални числа. Когато се извършва деление на целочислен с целочислен тип (например **int**, **long**, **sbyte**, …), върнатият резултат е отново целочислен (без закръгляне, с отрязване на дробната част). Такова деление се нарича целочислено. Например при целочислено деление 7 / 3 = 2. Целочислено деление на 0 не е позволено и при опит да бъде извършено, се получава грешка по време на изпълнение на програмата **DivideByZeroException**. Остатъкът от целочислено делене на цели числа може да се получи чрез оператора `%`. Например 7 % 3 = 1, а -10 % 2 = 0.

При деление на две реални числа или на две числа, от които едното е реално, се извършва реално делене (не целочислено) и резултатът е реално число с цяла и дробна част. Например 5.0 / 2 = 2.5. При делене на реални числа е позволено да се дели на 0.0 и резултатът е съответно **+∞**, **-∞** или **NaN**.

Операторът за увеличаване с единица (increment) **++** добавя единица към стойността на променливата, а съответно операторът **--** (decrement) изважда единица от стойността. Когато използваме операторите **++** и **--** като префикс (поставяме ги непосредствено преди променливата), първо се пресмята новата стойност, а после се връща резултата, докато при използването на операторите като постфикс (поставяме оператора непосредствено след променливата) първо се връща оригиналната стойност на операнда, а после се добавя или изважда единица към нея.

Аритметични оператори – примери

Ето няколко примера за аритметични оператори и тяхното действие:

```
int squarePerimeter = 17;
double squareSide = squarePerimeter / 4.0;
double squareArea = squareSide * squareSide;
Console.WriteLine(squareSide); // 4.25
Console.WriteLine(squareArea); // 18.0625
```

```
int a = 5;
int b = 4;
Console.WriteLine(a + b);      // 9
Console.WriteLine(a + b++);    // 9
Console.WriteLine(a + b);      // 10
Console.WriteLine(a + (++b));  // 11
Console.WriteLine(a + b);      // 11
Console.WriteLine(14 / a);     // 2
Console.WriteLine(14 % a);     // 4

int one = 1;
int zero = 0;
// Console.WriteLine(one / zero); // DivideByZeroException

double dMinusOne = -1.0;
double dZero = 0.0;
Console.WriteLine(dMinusOne / zero); // -Infinity
Console.WriteLine(one / dZero); // Infinity
```

Логически оператори

Логическите оператори приемат булеви стойности и връщат булев резултат (**`true`** или **`false`**). Основните булеви оператори са "И" (**`&&`**), "ИЛИ" (**`||`**), изключващо "ИЛИ" (**`^`**) и логическо отрицание (**`!`**).

Следва таблица с логическите оператори в C# и операциите, които те извършват:

x	**y**	**!x**	**x && y**	**x || y**	**x ^ y**
true	**true**	**false**	**true**	**true**	**false**
true	**false**	**false**	**false**	**true**	**true**
false	**true**	**true**	**false**	**true**	**true**
false	**false**	**true**	**false**	**false**	**false**

От таблицата, както и от следващия пример става ясно, че логическото "И" (&&) връща истина, само тогава, когато и двете променливи съдържат истина. Логическото "ИЛИ" (||) връща истина, когато поне един от операндите е истина. Операторът за логическо отрицание (!) сменя стойността на аргумента. Например, ако операндът е имала стойност **`true`** и приложим оператор за отрицание, новата стойност ще бъде **`false`**. Операторът за отрицание е едноаргументен и се слага пред аргумента. Изключващото "ИЛИ" (^) връща резултат **`true`**, когато само един от двата операнда има стойност **`true`**. Ако двата операнда имат различни стойности изключващото "ИЛИ" ще върне резултат **`true`**, ако имат еднакви стойности ще върне **`false`**.

Логически оператори – пример

Следва пример за използване на логически оператори, който илюстрира тяхното действие:

```
bool a = true;
bool b = false;
Console.WriteLine(a && b);                 // False
Console.WriteLine(a || b);                 // True
Console.WriteLine(!b);                     // True
Console.WriteLine(b || true);              // True
Console.WriteLine((5 > 7) ^ (a == b));     // False
```

Закони на Де Морган

Логическите операции се подчиняват на законите на Де Морган от математическата логика:

```
!(a && b) == (!a || !b)
!(a || b) == (!a && !b)
```

Първият закон твърди, че отрицанието на конюнкцията (логическо И) на две съждения е равна на дизюнкцията (логическо ИЛИ) на техните отрицания.

Вторият закон твърди, че отрицанието на дизюнкцията на две съждения е равно на конюнкцията на техните отрицания.

Оператор за съединяване на низове

Операторът **+** се използва за съединяване на символни низове (**string**). Той слепва два или повече низа и връща резултата като нов низ. Ако поне един от аргументите в израза е от тип **string**, и има други операнди, които не са от тип **string**, то те автоматично ще бъдат преобразувани към тип **string**.

Оператор за съединяване на низове – пример

Ето един пример, в който съединяваме няколко символни низа, както и стрингове с числа:

```
string csharp = "C#";
string dotnet = ".NET";
string csharpDotNet = csharp + dotnet;
Console.WriteLine(csharpDotNet); // C#.NET
string csharpDotNet4 = csharpDotNet + " " + 4;
Console.WriteLine(csharpDotNet4); // C#.NET 4
```

В примера инициализираме две променливи от тип **string** и им задаваме стойности. На третия и четвъртия ред съединяваме двата стринга и подаваме резултата на метода `Console.WriteLine()`, за да го отпечата на конзолата. На следващия ред съединяваме получения низ с интервал и числото 4. Върнатия резултат записваме в променливата `csharpDotNet4`, който автоматично ще бъде преобразуван към тип `string`. На последния ред подаваме резултата за отпечатване.

Конкатенацията (слепването на два низа) на стрингове е бавна операция и трябва да се използва внимателно. Препоръчва се използването на класа `StringBuilder` **при нужда от итеративни (повтарящи се) операции върху символни низове.**

В главата "Символни низове" ще обясним в детайли защо при операции над символни низове, изпълнени в цикъл, задължително трябва да се използва гореспоменатия клас `StringBuilder`.

Побитови оператори

Побитов оператор (**bitwise operator**) означава оператор, който действа над двоичното представяне на числовите типове. В компютрите всички данни и в частност числовите данни се представят като поредица от нули и единици. За целта се използва **двоичната бройна система**. Например числото `55` в двоична бройна система се представя като `00110111`.

Двоичното представяне на данните е удобно, тъй като нулата и единицата в електрониката могат да се реализират чрез логически схеми, в които нулата се представя като "няма ток" или примерно с напрежение -5V, а единицата се представя като "има ток" или примерно с напрежение +5V.

Ще разгледаме в дълбочина двоичната бройна система в главата "Бройни системи", а за момента можем да считаме, че числата в компютрите се представят като нули и единици и че побитовите оператори служат за анализиране и промяна на точно тези нули и единици.

Побитовите оператори много приличат на логическите. Всъщност можем да си представим, че логическите и побитовите оператори извършват едно и също нещо, но върху различни типове данни. Логическите оператори работят над стойностите `true` и `false` (булеви стойности), докато побитовите работят над числови стойности и се прилагат побитово над тяхното двоично представяне, т.е. работят върху битовете на числото (съставящите го цифри `0` и `1`). Също както при логическите оператори, в C# има оператори за побитово "И" (`&`), побитово "ИЛИ" (`|`), побитово отрицание (`~`) и изключващо "ИЛИ" (`^`).

Побитови оператори и тяхното действие

Действието на побитовите оператори над двоичните цифри 0 и 1 е показано в следната таблица:

x	y	~x	x & y	x | y	x ^ y
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

Както виждаме, побитовите и логическите оператори си приличат много. Разликата в изписването на "И" и "ИЛИ" е че при логическите оператори се пише двоен амперсанд (&&) и двойна вертикална черта (||), а при битовите – единични (& и |). Побитовият и логическият оператор за изключващо или е един и същ "^". За логическо отрицание се използва "`!`", докато за побитово отрицание (инвертиране) се използва "~".

В програмирането има още два побитови оператора, които нямат аналог при логическите. Това са побитовото изместване в ляво (<<) и побитовото изместване в дясно (>>). Използвани над числови стойности те преместват всички битове на стойността, съответно на ляво или надясно, като цифрите, излезли извън обхвата на числото, се губят и се заместват с 0.

Операторите за преместване се използват по следния начин: от ляво на оператора слагаме променливата (операндът), над която ще извършим операцията, вдясно на оператора поставяме число, указващо с колко знака искаме да отместим битовете. Например **`3 << 2`** означава, че искаме да преместим два пъти наляво битовете на числото 3. Числото 3 представено в битове изглежда така: `"0000 0011"`. Когато го преместим два пъти в ляво неговата двоична стойност ще изглежда така: `"0000 1100"`, а на тази поредица от битове отговаря числото 12. Ако се вгледаме в примера можем да забележим, че реално сме умножили числото по 4. Самото побитово преместване може да се представи като умножение (побитово преместване вляво) или делене (преместване в дясно) някаква степен на числото 2. Това явление е следствие от природата на двоичната бройна система. Пример за преместване надясно е **`6 >> 2`**, което означава да преместим двоичното число "`0000 0110`" с две позиции надясно. Това означава, че ще изгубим двете най-десни цифри и ще допълним с две нули отляво. Резултатът е "`0000 0001`", т.е. числото 1.

Побитови оператори – пример

Ето един пример за работа с побитови оператори. Двоичното представяне на числата и резултатите от различните оператори е дадено в коментари:

```
byte a = 3;                 // 0000 0011 = 3
byte b = 5;                 // 0000 0101 = 5

Console.WriteLine(a | b);   // 0000 0111 = 7
Console.WriteLine(a & b);   // 0000 0001 = 1
Console.WriteLine(a ^ b);   // 0000 0110 = 6
Console.WriteLine(~a & b);   // 0000 0100 = 4
Console.WriteLine(a << 1);   // 0000 0110 = 6
Console.WriteLine(a << 2);   // 0000 1100 = 12
Console.WriteLine(a >> 1);   // 0000 0001 = 1
```

В примера първо създаваме и инициализираме стойностите на две променливи **a** и **b**. След това отпечатваме на конзолата, резултатите от няколко побитови операции над двете променливи. Първата операция, която прилагаме е "ИЛИ". От примера се вижда, че за всички позиции, на които е имало 1 в двоичното представяне на променливите **a** и **b**, има 1 и в резултата. Втората операция е "И". Резултатът от операцията съдържа 1 само в най-десния бит, защото двете променливи имат едновременно 1 само в най-десния си бит. Изключващото "ИЛИ" връща единици само на позициите, където **a** и **b** имат различни стойности на двоичните си битове. След това в примера е илюстрирана работата на логическото отрицание и побитовото преместване вляво и вдясно.

Оператори за сравнение

Операторите за сравнение в C# се използват за сравняване на два или повече операнди. C# поддържа следните оператори за сравнение:

- по-голямо (>)
- по-малко (<)
- по-голямо или равно (>=)
- по-малко или равно (<=)
- равенство (==)
- различие (!=)

Всички оператори за сравнение в C# са двуаргументни (приемат два операнда), а върнатият от тях резултат е булев (**true** или **false**). Операторите за сравнение имат по-малък приоритет от аритметичните, но са с по-голям приоритет от операторите за присвояване на стойност.

Оператори за сравнение – пример

Следва пример, който демонстрира употребата на операторите за сравнение в C#:

```
int x = 10, y = 5;
Console.WriteLine("x > y : " + (x > y));    // True
Console.WriteLine("x < y : " + (x < y));    // False
Console.WriteLine("x >= y : " + (x >= y));  // True
Console.WriteLine("x <= y : " + (x <= y));  // False
Console.WriteLine("x == y : " + (x == y));  // False
Console.WriteLine("x != y : " + (x != y));  // True
```

В примерната програма, първо създаваме две променливи `x` и `y` и им присвояваме стойностите 10 и 5. На следващия ред отпечатваме на конзолата посредством метода `Console.WriteLine()` резултатът от сравняването на двете променливи `x` и `y` посредством оператора >. Върнатият резултат е `true`, защото `x` има по-голяма стойност от `y`. Аналогично в следващите редове се отпечатват резултатите от останалите 5 оператора за сравнение между променливите `x` и `y`.

Оператори за присвояване

Операторът за присвояване на стойност на променливите е "=" (символът равно). Синтаксисът, който се използва за присвояване на стойности, е следният:

```
операнд1 = литерал, израз или операнд2;
```

Оператори за присвояване – пример

Ето един пример, в който използваме оператора за присвояване на стойност:

```
int x = 6;
string helloString = "Здравей стринг.";
int y = x;
```

В горния пример присвояваме стойност 6 на променливата `x`. На втория ред присвояваме текстов литерал на променливата `helloString`, а на третия ред копираме стойността от променливата `x` в променливата `y`.

Каскадно присвояване

Операторът за присвояване може да се използва и каскадно (да се използва повече от веднъж в един и същ израз). В този случай присвояванията се извършват последователно отдясно наляво. Ето един пример:

```
int x, y, z;
x = y = z = 25;
```

На първия ред от примера създаваме три променливи, а на втория ред ги инициализираме със стойност 25.

Операторът за присвояване в C# е "=", докато операторът за сравнение е "==". Размяната на двата оператора е честа причина за грешки при писането на код. Внимавайте да не объркате оператора за сравнение с оператора за присвояване, тъй като те много си приличат.

Комбинирани оператори за присвояване

Освен оператора за присвояване в C# има и комбинирани оператори за присвояване. Те спомагат за съкращаване на обема на кода чрез изписване на две операции заедно с един оператор: операция и присвояване. Комбинираните оператори имат следния синтаксис:

```
операнд1 оператор = операнд2;
```

Горният израз е идентичен със следния:

```
операнда1 = операнд1 оператор операнд2;
```

Ето един пример за комбиниран оператор за присвояване:

```
int x = 2;
int y = 4;

x *= y; // Same as x = x * y;
Console.WriteLine(x); // 8
```

Най-често използваните комбинирани оператори за присвояване са **+=** (добавя стойността на **операнд2** към **операнд1**), **-=** (изважда стойността на операнда в дясно от стойността на тази в ляво). Други комбинирани оператори за присвояване са ***=**, **/=** и **%=**.

Следващият пример дава добра по-представа как работят комбинираните оператори за присвояване:

```
int x = 6;
int y = 4;

Console.WriteLine(y *= 2);   // 8
int z = y = 3;               // y=3 and z=3

Console.WriteLine(z);        // 3
Console.WriteLine(x |= 1);   // 7
Console.WriteLine(x += 3);   // 10
Console.WriteLine(x /= 2);   // 5
```

В примера първо създаваме променливите **x** и **y** и им присвояваме стойностите 6 и 4. На следващият ред принтираме на конзолата **y**, след като сме му присвоили нова стойност посредством оператора ***=** и литерала `2`. Резултатът от операцията е 8. По нататък в примера прилагаме други съставни оператори за присвояване и извеждаме получения резултат на конзолата.

Условен оператор ?:

Условният оператор `?:` използва булевата стойност от един израз, за да определи кой от други два израза да бъде пресметнат и върнат като резултат. Операторът работи над 3 операнда и за това се нарича тернарен. Символът "`?`" се поставя между първия и втория операнд, а "`:`" се поставя между втория и третия операнд. Първият операнд (или израз) трябва да е от булев тип, а другите два операнда трябва да са от един и същ тип, например числа или стрингове.

Синтаксисът на оператора `?:` е следният:

```
операнд1 ? операнд2 : операнд3
```

Той работи така: ако **`операнд1`** има стойност **`true`**, операторът връща като резултат **`операнд2`**. Иначе (ако **`операнд1`** има стойност **`false`**), операторът връща резултат **`операнд3`**.

По време на изпълнение се пресмята стойността на първия аргумент. Ако той има стойност **`true`**, тогава се пресмята втория (среден) аргумент и той се връща като резултат. Обаче, ако пресметнатият резултат от първия аргумент е **`false`**, то тогава се пресмята третият (последният) аргумент и той се връща като резултат.

Условен оператор ?: – пример

Ето един пример за употребата на оператора "`?:`":

```
int a = 6;
int b = 4;
Console.WriteLine(a > b ? "a>b" : "b<=a"); // a>b
int num = a == b ? 1 : -1; // num will have value -1
```

Други оператори

Досега разгледахме аритметичните оператори, логическите и побитовите оператори, оператора за конкатенация на символни низове, също и условния оператор `?:`. Освен тях в C# има още няколко оператора, на които си струва да обърнем внимание:

- Операторът за достъп "`.`" (точка) се използва за достъп до член променливите или методите на даден клас или обект. Пример за използването на оператора точка:

```
Console.WriteLine(DateTime.Now); // Prints the date + time
```

- Квадратни скоби `[]` се използват за достъп до елементите на масив по индекс и затова се нарича още индексатор. Индексатори се ползват още за достъп до символите в даден стринг. Пример:

```
int[] arr = { 1, 2, 3 };
Console.WriteLine(arr[0]); // 1
string str = "Hello";
Console.WriteLine(str[1]); // e
```

- Скоби **()** се използват за предефиниране приоритета на изпълнение на изразите и операторите. Вече видяхме как работят скобите.
- Операторът за преобразуване на типове **(type)** се използва за преобразуване на променлива от един тип в друг. Ще се запознаем с него в детайли в секцията "Преобразуване на типовете".
- Операторът **as** също се използва за преобразуване на типове, но при невалидност на преобразуването връща **null**, а не изключение.
- Операторът **new** се използва за създаването и инициализирането на нови обекти. Ще се запознаем в детайли с него в главата "Създаване и използване на обекти".
- Операторът **is** се използва за проверка дали даден обект е съвместим с даден тип.
- Операторът **??** е подобен на условния оператор **?:**. Разликата е, че той се поставя между два операнда и връща левия операнд само ако той няма стойност **null**, в противен случай връща десния. Пример:

```
int? a = 5;
Console.WriteLine(a ?? -1); // 5
string name = null;
Console.WriteLine(name ?? "(no name)"); // (no name)
```

Други оператори – примери

Ето няколко примера за операторите, които разгледахме в тази секция:

```
int a = 6;
int b = 3;

Console.WriteLine(a + b / 2);                        // 7
Console.WriteLine((a + b) / 2);                      // 4

string s = "Beer";
Console.WriteLine(s is string);                      // True

string notNullString = s;
string nullString = null;
Console.WriteLine(nullString ?? "Unspecified");   // Unspecified
Console.WriteLine(notNullString ?? "Specified"); // Beer
```

Преобразуване на типовете

По принцип операторите работят върху аргументи от еднакъв тип данни. Въпреки това в C# има голямо разнообразие от типове данни, от които можем да избираме най-подходящия за определена цел. За да извършим операция върху променливи от два различни типа данни ни се налага да преобразуваме двата типа към един и същ. Преобразуването на типовете (typecasting) бива явно и неявно (implicit typecasting и explicit typecasting).

Всички изрази в езика C# имат тип. Този тип може да бъде изведен от структурата на израза и типовете, променливите и литералите използвани в него. Възможно е да се напише израз, който е с неподходящ тип за конкретния контекст. В някой случаи това ще доведе до грешка при компилацията на програмата, но в други контекстът може да приеме тип, който е сходен или свързан с типа на израза. В този случай програмата извършва скрито преобразуване на типове.

Специфично преобразуване от тип **`S`** към тип **`T`** позволя на израза от тип **`S`** да се третира като израз от тип **`T`** по време на изпълнението на програмата. В някои случай това ще изисква проверка на валидността на преобразуването. Ето няколко примера:

- Преобразуване от тип **`object`** към тип **`string`** ще изисква проверка по време на изпълнение, за да потвърди, че стойността е наистина инстанция от тип **`string`**.
- Преобразуване от тип **`string`** към **`object`** не изисква проверка. Типът **`string`** е наследник на типа **`object`** и може да бъде преобразуван към базовия си клас без опасност от грешка или загуба на данни. На наследяването ще се спрем в детайли в главата "Принципи на обектно-ориентираното програмиране".
- Преобразуване от тип **`int`** към **`long`** може да се извърши без проверка по време на изпълнението, защото няма опасност от загуба на данни, тъй като множеството от стойности на типа **`long`** е подмножество на стойностите на типа **`int`**.
- Преобразуване от тип **`double`** към **`long`** изисква преобразуване от 64-битова плаваща стойност към 64-битова целочислена. В зависимост от стойността, може да се получи загуба на данни и поради това е необходимо изрично преобразуване на типовете.

В C# не всички типове могат да бъдат преобразувани във всички други, а само към някои определени. За удобство ще групираме някой от възможните преобразувания в C# според вида им в две категории:

- скрито (неявно) преобразуване;
- изрично (явно) преобразуване;
- преобразуване от и към **`string`**.

Неявно (implicit) преобразуване на типове

Неявното (скритото) преобразуване на типове е възможно единствено, когато няма възможност от загуба на данни при преобразуването, т.е. когато конвертираме от тип с по-малък обхват към тип с по-голям обхват (примерно от **`int`** към **`long`**). За да направим неявно преобразуване не е нужно да използваме какъвто и да е оператор и затова такова преобразуване се нарича още скрито (implicit). Преобразуването става автоматично от компилатора, когато присвояваме стойност от по-малък обхват в променлива с по-голям обхват или когато в израза има няколко типа с различен обхват. Тогава преобразуването става към типа с най-голям обхват.

Неявно преобразуване на типове – пример

Ето един пример за неявно (implicit) преобразуване на типове:

```
int myInt = 5;
Console.WriteLine(myInt); // 5

long myLong = myInt;
Console.WriteLine(myLong); // 5

Console.WriteLine(myLong + myInt); // 10
```

В примера създаваме променлива **`myInt`** от тип **`int`** и присвояваме стойност 5. По-надолу създаваме променлива **`myLong`** от тип **`long`** и задаваме стойността, съдържаща се в **`myInt`**. Стойността запазена в **`myLong`**, автоматично се конвертира от тип **`int`** към тип **`long`**. Накрая в примера извеждаме резултата от събирането на двете променливи. Понеже променливите са от различен тип, те автоматично се преобразуват към типа с по-голям обхват, тоест към **`long`** и върнатият резултат, който се отпечатва на конзолата, отново е **`long`**. Всъщност подадения параметър на метода **`Console.WriteLine()`** е от тип **`long`**, но вътре в метода той отново ще бъде конвертиран, този път към тип **`string`**, за да може да бъде отпечатан на конзолата. Това преобразование се извършва чрез метода **`Long.ToString()`**.

Възможни неявни преобразования

Ето някои от възможните неявни (implicit) преобразувания на примитивни типове в C#:

- **`sbyte`** → **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, **`decimal`**;
- **`byte`** → **`short`**, **`ushort`**, **`int`**, **`uint`**, **`long`**, **`ulong`**, **`float`**, **`double`**, **`decimal`**;
- **`short`** → **`int`**, **`long`**, **`float`**, **`double`**, **`decimal`**;
- **`ushort`** → **`int`**, **`uint`**, **`long`**, **`ulong`**, **`float`**, **`double`**, **`decimal`**;

- **char** → **ushort**, **int**, **uint**, **long**, **ulong**, **float**, **double**, **decimal** (въпреки, че **char** е символен тип, в някои случаи той може да се разглежда като число и има поведение на числов тип, дори може да участва в числови изрази);
- **uint** → **long**, **ulong**, **float**, **double**, **decimal**;
- **int** → **long**, **float**, **double**, **decimal**;
- **long** → **float**, **double**, **decimal**;
- **ulong** → **float**, **double**, **decimal**;
- **float** → **double**.

При преобразуването на типове от по-малък обхват към по-голям няма загуба на данни. Числовата стойност остава същата след преобразуването. Както във всяко правило и тук има малко изключение. Когато преобразуваме тип **int** към тип **float** (32-битови стойности), разликата е, че **int** използва всичките си битове за представяне на едно целочислено число, докато **float** използва част от битовете си за представянето на плаващата запетая. Оттук следва, че е възможно при преобразуване от **int** към **float** да има загуба на точност, поради закръгляне. Същото се отнася и за преобразуването на 64-битовия **long** към 64-битовия **double**.

Изрично (explicit) преобразуване на типове

Изричното преобразуване на типове (explicit typecasting) се използва винаги, когато има вероятност за загуба на данни. Когато конвертираме тип с плаваща запетая към целочислен тип, винаги има загуба на данни, идваща от премахването на дробната част и е задължително използването на изрично преобразуване (например **double** към **long**). За да направим такова конвертиране е нужно изрично да използваме оператора за преобразуване на данни (**type**). Възможно е да има загуба на данни също, когато конвертираме от тип с по-голям обхват към тип с по-малък (**double** към **float** или **long** към **int**).

Изрично преобразуване на типове – пример

Следният пример илюстрира употребата на изрично конвертиране на типовете и загуба на данни, която може да настъпи в някои случаи:

```
double myDouble = 5.1d;
Console.WriteLine(myDouble); // 5.1

long myLong = (long)myDouble;
Console.WriteLine(myLong); // 5

myDouble = 5e9d; // 5 * 10^9
Console.WriteLine(myDouble); // 5000000000

int myInt = (int)myDouble;
```

```
Console.WriteLine(myInt); // -2147483648
Console.WriteLine(int.MinValue); // -2147483648
```

На първия ред от примера присвояваме стойността 5.1 на променливата **`myDouble`**. След като я преобразуваме (изрично), посредством оператора **`(long)`** към тип **`long`** и изкараме на конзолата променливата **`myLong`**, виждаме, че променливата е изгубила дробната си част, защото **`long`** е целочислен тип. След това присвояваме на реалната променлива с двойна точност **`myDouble`** стойност 5 милиарда. Накрая конвертираме **`myDouble`** към **`int`** посредством оператора **`(int)`** и отпечатваме променливата **`myInt`**. Резултатът е същия, както и когато отпечатаме **`int.MinValue`**, защото **`myDouble`** съдържа в себе си по-голяма стойност от обхвата на **`int`**.

Не винаги е възможно да се предвиди каква ще бъде стойността на дадена променлива след препълване на обхвата и! Затова използвайте достатъчно големи типове и внимавайте при преминаване към "по-малък" тип.

Загуба на данни при преобразуване на типовете

Ще дадем още един пример за загуба на данни при преобразуване на типове:

```
long myLong = long.MaxValue;
int myInt = (int)myLong;

Console.WriteLine(myLong); // 9223372036854775807
Console.WriteLine(myInt); // -1
```

Операторът за преобразуване може да се използва и при неявно преобразуване по-желание. Това допринася за четимостта на кода, намалява шанса за грешки и се счита за добра практика от много програмисти.

Ето още няколко примера за преобразуване на типове:

```
float heightInMeters = 1.74f; // Explicit conversion
double maxHeight = heightInMeters; // Implicit
double minHeight = (double)heightInMeters; // Explicit
float actualHeight = (float)maxHeight; // Explicit

float maxHeightFloat = maxHeight; // Compilation error!
```

В примера на последния ред имаме израз, който ще генерира грешка при компилирането. Това е така, защото се опитваме да конвертираме неявно от тип **`double`** към тип **`float`**, от което може да има загуба на данни. C# е строго типизиран език за програмиране и не позволява такъв вид присвояване на стойности.

Прихващане на грешки при преобразуване на типовете

Понякога е удобно вместо да получаваме грешен резултат при евентуално препълване при преминаване от по-голям към по-малък тип, да получим уведомление за проблема. Това става чрез ключовата дума `checked`, която включва уведомлението за препълване при целочислените типове:

```
double d = 5e9d; // 5 * 10^9
Console.WriteLine(d); // 5000000000
int i = checked((int)d); // System.OverflowException
Console.WriteLine(i);
```

При изпълнението на горния фрагмент от код се получава изключение (т.е. уведомление за грешка) `OverflowException`. Повече за изключенията и средствата за тяхното прихващане и обработка можете да прочетете в главата "Обработка на изключения".

Възможни изрични преобразования

Явните (изрични) преобразувания между числовите типове в езика C# са възможни между всяка двойка измежду следните типове:

```
sbyte, byte, short, ushort, char, int, uint, long, ulong, float,
double, decimal
```

При тези преобразувания могат да се изгубят, както данни за големината на числото, така и информация за неговата точност (precision).

Забележете, че преобразуването към `string` и от `string` не е възможно да се извършва чрез преобразуване на типовете (typecasting).

Преобразуване към символен низ

При необходимост можем да преобразуваме към низ, всеки отделен тип, включително и стойността `null`. Преобразуването на символни низове става автоматично винаги, когато използваме оператора за конкатенация (+) и някой от аргументите не е от тип низ. В този случай аргумента се преобразува към низ и операторът връща нов низ представляващ конкатенацията на двата низа.

Друг начин да преобразуваме различни обекти към тип символен низ е като извикаме метода `ToString()` на съответната променлива или стойност. Той е валиден за всички типове данни в .NET Framework. Дори извикването `3.ToString()` е напълно валидно в C# и като резултат ще се върне низа "3".

Преобразуване към символен низ – пример

Нека разгледаме няколко примера за преобразуване на различни типове данни към символен низ:

```
int a = 5;
int b = 7;

string sum = "Sum=" + (a + b);
Console.WriteLine(sum);

String incorrect = "Sum=" + a + b;
Console.WriteLine(incorrect);

Console.WriteLine(
  "Perimeter = " + 2 * (a + b) + ". Area = " + (a * b) + ".");
```

Резултатът от изпълнението на примера е следният:

```
Sum=12
Sum=57
Perimeter = 24. Area = 35.
```

От резултата се вижда, че долепването на число към символен низ връща като резултата символния низ, следван от текстовото представяне на числото. Забележете, че операторът "+" за залепване на низове може да предизвика неприятен ефект при събиране на числа, защото има еднакъв приоритет с оператора "+" за събиране. Освен, ако изрично не променим приоритета на операциите чрез поставяне на скоби, те винаги се изпълняват отляво надясно.

Повече подробности по въпроса как да преобразуваме от и към **`string`** ще разгледаме в главата "Вход и изход от конзолата".

Изрази

Голяма част от работата на една програма е пресмятането на изрази. Изразите представляват поредици от оператори, литерали и променливи, които се изчисляват до определена стойност от някакъв тип (число, символен низ, обект или друг тип). Ето няколко примера за изрази:

```
int r = (150-20) / 2 + 5;

// Expression for calculation of the surface of the circle
double surface = Math.PI * r * r;

// Expression for calculation of the perimeter of the circle
double perimeter = 2 * Math.PI * r;

Console.WriteLine(r);
Console.WriteLine(surface);
Console.WriteLine(perimeter);
```

В примера са дефинирани три израза. Първият израз пресмята радиуса на дадена окръжност. Вторият пресмята площта на окръжността, а последният намира периметърът ѝ. Ето какъв е резултатът е изпълнения на горния програмен фрагмент:

```
70
15393,80400259
439,822971502571
```

Изчисляването на израз може да има и странични действия, защото изразът може да съдържа вградени оператори за присвояване, увеличаване или намаляване на стойност (increment, decrement) и извикване на методи. Ето пример за такъв страничен ефект:

```
int a = 5;
int b = ++a;

Console.WriteLine(a); // 6
Console.WriteLine(b); // 6
```

При съставянето на изрази трябва да се имат предвид типовете данни и поведението на използваните оператори. Пренебрегването на тези особености може да доведе до неочаквани резултати. Ето един прост пример:

```
double d = 1 / 2;
Console.WriteLine(d); // 0, not 0.5

double half = (double)1 / 2;
Console.WriteLine(half); // 0.5
```

В примера се използва израз, който разделя две цели числа и присвоява резултата на променлива от тип **`double`**. Резултатът за някои може да е неочакван, но това е защото игнорират факта, че операторът "/" за цели числа работи целочислено и резултатът е цяло число, получено чрез отрязване на дробната част.

От примера се вижда още, че ако искаме да извършим деление с резултат дробно число, е необходимо да преобразуваме до **`float`** или **`double`** поне един от операндите. При този сценарий делението вече не е целочислено и резултатът е коректен.

Друг интересен пример е делението на 0. Повечето програмисти си мислят, че делението на 0 е невалидна операция и предизвиква грешка по време на изпълнение (exception), но това всъщност е вярно само за целочисленото деление на 0. Ето един пример, който показва, че при нецелочислено деление на 0 се получава резултат Infinity или NaN:

```
int num = 1;
double denum = 0; // The value is 0.0 (real number)
int zeroInt = (int) denum; // The value is 0 (integer number)
Console.WriteLine(num / denum); // Infinity
Console.WriteLine(denum / denum); // NaN
Console.WriteLine(zeroInt / zeroInt); // DivideByZeroException
```

При работата с изрази е важно да се използват скоби винаги, когато има и най-леко съмнение за приоритетите на използваните операции. Ето един пример, който показва колко са полезни скобите:

```
double incorrect = (double)((1 + 2) / 4);
Console.WriteLine(incorrect); // 0

double correct = ((double)(1 + 2)) / 4;
Console.WriteLine(correct); // 0.75

Console.WriteLine("2 + 3 = " + 2 + 3); // 2 + 3 = 23
Console.WriteLine("2 + 3 = " + (2 + 3)); // 2 + 3 = 5
```

Упражнения

1. Напишете израз, който да проверява дали дадено цяло число е четно или нечетно.
2. Напишете булев израз, който да проверява дали дадено цяло число се дели на 5 и на 7 без остатък.
3. Напишете израз, който да проверява дали третата цифра (отдясно на ляво) на дадено цяло число е 7.
4. Напишете израз, който да проверява дали третия бит на дадено число е 1 или 0.
5. Напишете израз, който изчислява площта на трапец по дадени a, b и h.
6. Напишете програма, която за подадени от потребителя дължина и височина на правоъгълник, пресмята и отпечатва на конзолата неговия периметър и лице.
7. Силата на гравитационното поле на Луната е приблизително 17% от това на Земята. Напишете програма, която да изчислява тежестта на човек на Луната, по дадената тежест на Земята.
8. Напишете програма, която проверява дали дадена точка О (x, y) е вътре в окръжността К ((0,0), 5). Пояснение: точката (0,0) е център на окръжността, а радиусът й е 5.
9. Напишете програма, която проверява дали дадена точка О (x, y) е вътре в окръжността К ((0,0), 5) и едновременно с това извън правоъгълника ((-1, 1), (5, 5). Пояснение: правоъгълникът е зададен чрез координатите на горния си ляв и долния си десен ъгъл.

10. Напишете програма, която приема за вход четирицифрено число във формат *abcd* (например числото 2011) и след това извършва следните действия върху него:
    - Пресмята сбора от цифрите на числото (за нашия пример 2+0+1+1 = 4).
    - Разпечатва на конзолата цифрите в обратен ред: *dcba* (за нашия пример резултатът е 1102).
    - Поставя последната цифра, на първо място: *dabc* (за нашия пример резултатът е 1201).
    - Разменя мястото на втората и третата цифра: *acbd* (за нашия пример резултатът е 2101).
11. Дадено е число `n` и позиция `p`. Напишете поредица от операции, които да отпечатат стойността на бита на позиция `p` от числото `n` (0 или 1). Пример: n=35, p=5 -> 1. Още един пример: n=35, p=6 -> 0.
12. Напишете булев израз, който проверява дали битът на позиция `p` на цялото число `v` има стойност 1. Пример v=5, p=1 -> false.
13. Дадено е число n, стойност `v` (`v` = 0 или 1) и позиция `p`. Напишете поредица от операции, които да променят стойността на n, така че битът на позиция p да има стойност v. Пример n=35, p=5, v=0 -> n=3. Още един пример: n=35, p=2, v=1 -> n=39.
14. Напишете програма, която проверява дали дадено число `n` (`1 < n < 100`) е просто (т.е. се дели без остатък само на себе си и на единица).
15. * Напишете програма, която разменя стойностите на битовете на позиции 3, 4 и 5 с битовете на позиции 24, 25 и 26 на дадено цяло положително число.
16. * Напишете програма, която разменя битовете на позиции {p, p+1, …, p+k-1) с битовете на позиции {q, q+1, …, q+k-1} на дадено цяло положително число.

Решения и упътвания

1. Вземете остатъкът от делението на числото на 2 и проверете дали е 0 или 1 (съответно числото е четно или нечетно). Използвайте оператора `%` за пресмятане на остатък от целочислено деление.
2. Ползвайте логическо "И" (оператора `&&`) и операцията `%` за остатък при деление. Можете да решите задачата и чрез само една проверка – за деление на 35 (помислете защо).
3. Разделете числото на 100 и го запишете в нова променлива. Нея разделете на 10 и вземете остатъкът. Остатъкът от делението на 10 е третата цифра от първоначалното число. Проверете равна ли е на 7.

4. Използвайте побитово "И" върху числото и число, което има 1 само в третия си бит (т.е. числото 8, ако броенето на битовете започне от 0). Ако върнатият резултат е различен от 0, то третия бит е 1.

5. Формула за лице на трапец: **`S = (a + b) / 2 * h`**.

6. Потърсете в Интернет как се въвеждат цели числа от конзолата и използвайте формулата за лице на правоъгълник. Ако се затруднявате погледнете упътването на следващата задача.

7. Използвайте следния код, за да прочетете число от конзолата, след което го умножете по 0.17 и го отпечатайте:

```
Console.Write("Enter number: ");
int number = Convert.ToInt32(Console.ReadLine());
```

8. Използвайте питагоровата теорема $a^2 + b^2 = c^2$. За да е точката вътре в кръга, то **`x*x + y*y`** следва да е по-малко или равно на **5**.

9. Използвайте кода от предходната задача и добавете проверка за правоъгълника. Една точка е вътре в даден правоъгълник със стени успоредни на координатните оси, когато е вдясно от лявата му стена, вляво от дясната му стена, надолу от горната му стена и нагоре от долната му стена.

10. За да вземете отделните цифри на числото, можете да го делите на 10 и да взимате остатъка при деление на 10 последователно 4 пъти.

11. Ползвайте побитови операции:

```
int n = 35; // 00100011
int p = 6;
int i = 1; // 00000001
int mask = i << p; // Move the 1st bit left by p positions

// If i & mask are positive then the p-th bit of n is 1
Console.WriteLine((n & mask) != 0 ? 1 : 0);
```

12. Задачата е аналогична на предната.

13. Ползвайте побитови операции, по аналогия със задача 11. Можете да нулирате бита на позиция **`p`** в числото **`n`** по следния начин:

```
n = n & (~(1 << p));
```

Можете да установите в единица бита на позиция **`p`** в числото **`n`** по следния начин:

```
n = n | (1 << p);
```

Помислете как можете да комбинирате тези две упътвания.

14. Прочетете за цикли в Интернет. Използвайте цикъл и проверете числото за делимост на всички числа от 1 до корен квадратен от числото. В конкретната задача, тъй като ограничението е само до 100, можете предварително да намерите простите числа от 1 до 100 и да направите проверки дали даденото число **`n`** е равно на някое от тях.

15. За решението на тази задача използвайте комбинация от задачите за взимане и установяване на бит на определена позиция.

16. Използвайте предната задача и прочетете в интернет как се работи с цикли и масиви (в които да запишете битовете).

Telerik Academy

Присъедини се към Академията на Телерик!

АКАДЕМИЯТА НА ТЕЛЕРИК предоставя безплатно практическо обучение, насочено към всички млади хора, желаещи да станат умели .NET софтуерни инженери и да се присъединят към екипа на Телерик.

Академията подготвя бъдещите софтуерни специалисти за сложността, разнообразието и динамиката на днешната ИТ действителност, като за целта всички курсисти се обучават да работят с най-новите технологии на Майкрософт и черпят знания и опит от доказали се в областта на програмирането лектори и разработчици на софтуер.

В академията ще получите задълбочени знания и опит, изучавайки:

C#, .NET технологии, бази данни, SQL, Silverlight, WPF, ASP.NET, HTML5, разработка на мобилни приложения за iOS, Android и Windows Phone, основите на софтуерното инженерство

Академията на Телерик ви дава възможност да:

- **Учите напълно БЕЗПЛАТНО**
- **Изберете сред редица РАЗЛИЧНИ КУРСОВЕ**
- **Овладеете ОСНОВИТЕ на софтуерното инженерство**
- **Усвоите ПРОЦЕСА за разработка на софтуер**
- **Получите задълбочени теоретични и практически ИТ ПОЗНАНИЯ**
- **Станете умел .NET СОФТУЕРЕН ИНЖЕНЕР**
- **Започнете своята ИТ кариера в ТЕЛЕРИК – РАБОТОДАТЕЛ #1 в България за 2010 г.**

Само в рамките на две години **АКАДЕМИЯТА НА ТЕЛЕРИК** за софтуерни инженери успя да се наложи като безспорен лидер у нас в предлагането на допълнително обучение за софтуерни специалисти, спомагайки за успешния старт в кариерното развитие на стотици ентусиазирани младежи.

academy.telerik.com

academy@telerik.com *facebook.com/TelerikAcademy*

telerik
deliver more than expected

Глава 4. Вход и изход от конзолата

В тази тема...

В настоящата тема ще се запознаем с конзолата като средство за въвеждане и извеждане на данни. Ще обясним какво представлява тя, кога и как се използва, какви са принципите на повечето програмни езици за достъп до конзолата. Ще се запознаем с някои от възможностите на C# за взаимодействие с потребителя. Ще разгледаме основните потоци за входно-изходни операции **`Console.In`**, **`Console.Out`** и **`Console.Error`**, класът **`Console`** и използването на форматиращи низове за отпечатване на данни в различни формати.

Какво представлява конзолата?

Конзолата представлява прозорец на операционната система, през който потребителите могат да си взаимодействат със системните програми на операционната система или с други конзолни приложения. Взаимодействието се състои във въвеждане на текст от стандартния вход (най-често клавиатурата) или извеждане на текст на стандартния изход (най-често на екрана на компютъра). Тези действия са известни още, като **входно-изходни операции**. Текстът, изписван на конзолата носи определена информация и представлява поредица от символи изпратени от една или няколко програми.

За всяко конзолно приложение операционната система свързва устройства за вход и изход. По подразбиране това са клавиатурата и екрана, но те могат да бъдат пренасочвани към файл или други устройства.

Комуникация между потребителя и програмата

Голяма част от програмите си комуникират по някакъв начин с потребителя. Това е необходимо, за да може потребителя да даде своите инструкции към тях. Съвременните начини за комуникация са много и различни: те могат да бъдат през графичен или уеб-базиран интерфейс, конзола или други. Както споменахме, едно от средствата за комуникация между програмите и потребителя е конзолата, но тя става все по-рядко използвана. Това е така, понеже съвременните средства за реализация на потребителски интерфейс са по-удобни и интуитивни за работа.

Кога да използваме конзолата?

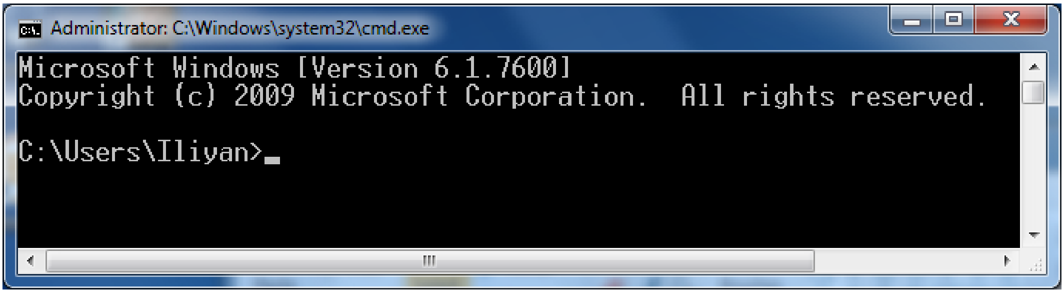
В някои случаи, конзолата си остава незаменимо средство за комуникация с потребителя. Един от тези случаи е при писане на малки и прости програмки, където е необходимо вниманието да е насочено към конкретния проблем, който решаваме, а не към елегантно представяне на резултата на потребителя. Тогава се използва просто решение за въвеждане или извеждане на резултат, каквото е конзолният вход-изход. Друг случай на употреба е когато искаме да тестваме малка част от кода на по-голямо приложение. Поради простотата на работа на конзолното приложение, можем да изолираме тази част от кода лесно и удобно, без да се налага да преминаваме през сложен потребителски интерфейс и поредица от екрани, за да стигнем до желания код за тестване.

Как да стартираме конзолата?

Всяка операционна система си има собствен начин за стартиране на конзолата. Под Windows например стартирането става по следния начин:

```
Start -> (All) Programs -> Accessories -> Command Prompt
```

След стартиране на конзолата, трябва да се появи черен прозорец, който изглежда по следния начин:



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Iliyan>_
```

При стартиране на конзолата, за текуща директория се използва личната директория на текущия потребител, която се извежда като ориентир за потребителя.

Конзолата може да се стартира и чрез последователността Start -> Run… -> пишем "cmd" в диалога и натискаме [Enter].

За по-добра визуализация на резултатите от сега нататък в тази глава вместо снимка на екрана (screenshot) от конзолата ще използваме вида:

```
Results from console
```

Подробно за конзолите

Системната конзола, още наричана "**`Command Prompt`**" или "**`shell`**" или "команден интерпретатор" е програма на операционната система, която осигурява достъп до системни команди, както и до голям набор програми, които са част от операционната система или са допълнително инсталирани към нея.

Думата "**`shell`**" (шел) означава "обвивка" и носи смисъла на обвивка между потребителя и вътрешността на операционната система.

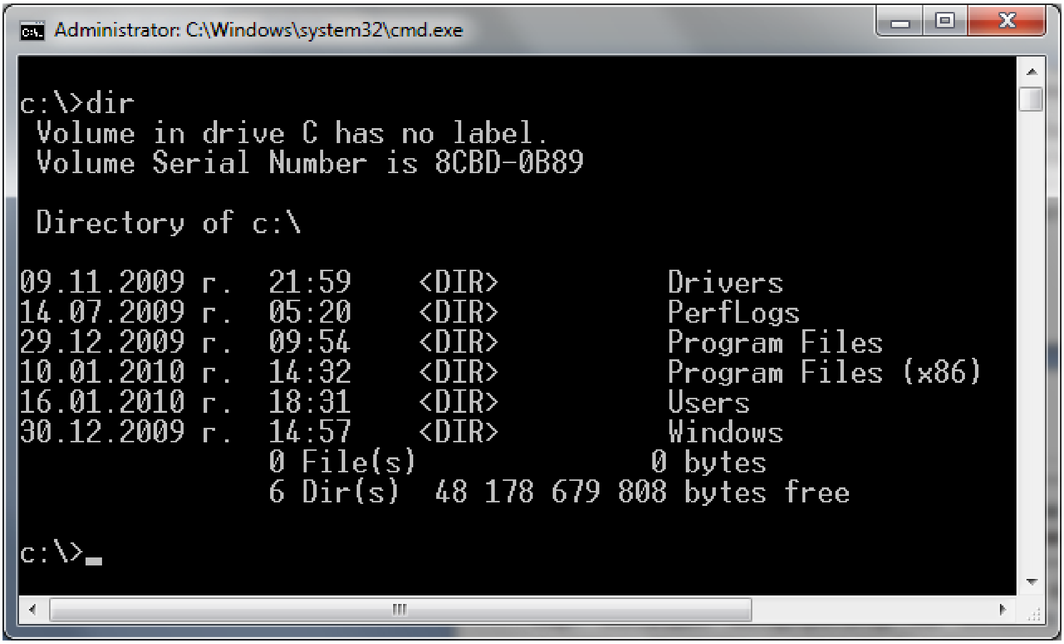
Така наречените "обвивки", могат да се разгледат в две основни категории, според това какъв интерфейс могат да предоставят към операционната система:

- Команден интерфейс (CLI – Command Line Interface) – представлява конзола за команди (като например **`cmd.exe`** в Windows и **`bash`** в Linux).
- Графичен интерфейс (GUI – Graphical User Interface) – представлява графична среда за работа (като например Windows Explorer).

И при двата вида, основната цел на обвивката е да стартира други програми, с които потребителят работи, макар че повечето интерпретатори поддържат и разширени функционалности, като например възможност за разглеждане съдържанието на директориите с файлове.

⚠	**Всяка операционна система има свой команден интерпретатор, който дефинира собствени команди.**

Например при стартиране на конзолата на Windows в нея се изпълнява т. нар. команден интерпретатор на Windows (**`cmd.exe`**), който изпълнява системни програми и команди в интерактивен режим. Например командата **`dir`**, показва файловете в текущата директория:



```
c:\>dir
 Volume in drive C has no label.
 Volume Serial Number is 8CBD-0B89

 Directory of c:\

09.11.2009 г.  21:59    <DIR>          Drivers
14.07.2009 г.  05:20    <DIR>          PerfLogs
29.12.2009 г.  09:54    <DIR>          Program Files
10.01.2010 г.  14:32    <DIR>          Program Files (x86)
16.01.2010 г.  18:31    <DIR>          Users
30.12.2009 г.  14:57    <DIR>          Windows
               0 File(s)              0 bytes
               6 Dir(s)  48 178 679 808 bytes free

c:\>_
```

Основни конзолни команди

Ще разгледаме някои базови конзолни команди, които ще са ни от полза при намиране и стартиране на програми.

Конзолни команди под Windows

Командният интерпретатор (конзолата) се нарича "**`Command Prompt`**" или "**`MS-DOS Prompt`**" (в по-старите версии на Windows). Ще разгледаме няколко базови команди за този интерпретатор:

Команда	**Описание**
`dir`	Показва съдържанието на текущата директория.
`cd <directory name>`	Променя текущата директория.

mkdir <directory name>	Създава нова директория в текущата.
rmdir <directory name>	Изтрива съществуваща директория.
type <file name>	Отпечатва съдържанието на файл.
copy <src file> <destination file>	Копира един файл в друг файл.

Ето пример за изпълнение на няколко команди в командния интерпретатор на Windows. Резултатът от изпълнението на командите се визуализира в конзолата:

```
C:\Documents and Settings\User1>cd "D:\Project2009\C# Book"

C:\Documents and Settings\User1>D:

D:\Project2008\C# Book>dir
 Volume in drive D has no label.
 Volume Serial Number is B43A-B0D6

 Directory of D:\Project2009\C# Book

26.12.2009 г.  12:24    <DIR>          .
26.12.2009 г.  12:24    <DIR>          ..
26.12.2009 г.  12:23           537 600 Chapter-4-Console-Input-
Output.doc
26.12.2009 г.  12:23    <DIR>          Test Folder
26.12.2009 г.  12:24                 0 Test.txt
               2 File(s)        537 600 bytes
               3 Dir(s)  24 154 062 848 bytes free

D:\Project2009\C# Book>
```

Стандартен вход-изход

Стандартният вход-изход известен още, като "**Standard I/O**" е системен входно-изходен механизъм създаден още от времето на Unix операционните системи. За вход и изход се използват специални периферни устройства, чрез които може да се въвеждат и извеждат данни.

Когато програмата е в режим на приемане на информация и очаква действие от страна на потребителя, в конзолата започва да мига курсор, подсказващ, че системата очаква въвеждане на команда.

По-нататък ще видим как можем да пишем C# програми, които очакват въвеждане на входни данни от конзолата.

Печатане на конзолата

В повечето програмни езици отпечатването и четенето на информация от конзолата е реализирано по различен начин, но повечето решения се базират на концепцията за "стандартен вход" и "стандартен изход".

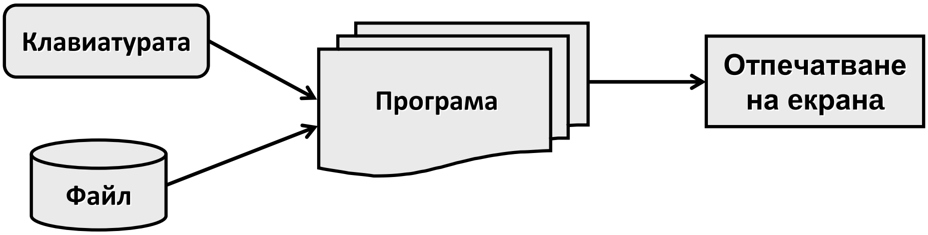
Стандартен вход и стандартен изход

Операционната система е длъжна да дефинира стандартни входно-изходни механизми за взаимодействие с потребителя. При стартиране на дадена конзолна програма, служебен код изпълняван в началото на програмата е отговорен за отварянето (затварянето) на потоци, към предоставените от операционната система механизми за вход-изход. Този служебен код инициализира програмната абстракция за взаимодействие с потребителя, заложена в съответния език за програмиране. По този начин стартираното приложение може да чете наготово потребителски вход от стандартния входен поток (в C# това е `Console.In`), може да записва информация в стандартния изходен поток (в C# това е `Console.Out`) и може да съобщава проблемни ситуации в стандартния поток за грешки (в C# това е `Console.Error`).

Концепцията за потоците ще бъде подробно разгледана по-късно. Засега ще се съсредоточим върху теоретичната основа, засягаща програмния вход и изход в C#.

Устройства за конзолен вход и изход

Освен от клавиатура, входът в едно приложение може да идва от много други места, като например файл, микрофон, бар-код четец и др. Изходът от една програма може да е на конзолата (на екрана), както и във файл или друго изходно устройство, например принтер:



Ще покажем базов пример онагледяващ отпечатването на текст в конзолата чрез абстракцията за достъп до стандартния вход и стандартния изход, предоставена ни от C#:

```
Console.Out.WriteLine("Hello World");
```

Резултатът от изпълнението на горния код би могъл да е следният:

```
Hello World
```

Потокът Console.Out

Класът **`System.Console`** има различни свойства и методи (класовете се разглеждат подробно в главата "Създаване и използване на обекти"), които се използват за четене и извеждане на текст на конзолата както и за неговото форматиране. Сред тях правят впечатление три свойства, свързани с въвеждането и извеждането на данни, а именно **`Console.Out`**, **`Console.In`** и **`Console.Error`**. Те дават достъп до стандартните потоци за отпечатване на конзолата, за четене от конзолата и до потока за съобщения за грешки съответно. Макар да бихме могли да ги използваме директно, другите методи на **`System.Console`** ни дават удобство на работа при входно/изходни операции на конзолата и реално най-често тези свойства се пренебрегват. Въпреки това е хубаво да не забравяме, че част от функционалността на конзолата работи върху тези потоци. Ако желаем, бихме могли да подменим потоците, като използваме съответно методите **`Console.SetOut(…)`**, **`Console.SetIn(…)`** и **`Console.SetError(…)`**.

Сега ще разгледаме най-често използваните методи за отпечатване на текст на конзолата.

Използване на Console.Write(…) и Console.WriteLine(…)

Работата със съответните методи е лесна, понеже може да се отпечатват всички основни типове (стринг, числени и примитивни типове):

Ето някой примери за отпечатването на различни типове данни:

```
// Print String
Console.WriteLine("Hello World");

// Print int
Console.WriteLine(5);

// Print double
Console.WriteLine(3.14159265358979);
```

Резултатът от изпълнението на този код изглежда така:

```
Hello World
5
3,14159265358979
```

Както виждаме, чрез **`Console.WriteLine(…)`** е възможно да отпечатаме различни типове данни, понеже за всеки от типовете има предефинирана версия на метода **`WriteLine(…)`** в класа **`Console`**.

Разликата между **`Write(…)`** и **`WriteLine(…)`**, е че методът **`Write(…)`** отпечатва в конзолата това, което му е подадено между скобите, но не прави нищо допълнително, докато методът **`WriteLine(…)`** в превод означава "отпечатай линия". Този метод прави това, което прави **`Write(…)`**, но в допълнение преминава на нов ред. В действителност методът не отпечатва нов ред, а просто слага "команда" за преместване на курсора на позицията, където започва новият ред.

Ето един пример, който илюстрира разликата между **`Write(…)`** и **`WriteLine(…)`**:

```
Console.WriteLine("I love");
Console.Write("this ");
Console.Write("Book!");
```

Изходът от този пример е:

```
I love
this Book!
```

Забелязваме, че изходът от примера е отпечатан на два реда, независимо че кодът е на три. Това се случва, понеже на първия ред от кода използваме **`WriteLine(…)`**, който отпечатва "**`I love`**" и след това се минава на нов ред. В следващите два реда от кода се използва методът **`Write(…)`**, който печата, без да минава на нов ред и по този начин думите "**`this`**" и "**`Book!`**" си остават на един и същи ред.

Конкатенация на стрингове

В общия случай C# не позволява използването на оператори върху стрингови обекти. Единственото изключение на това правило е операторът за събиране (+), който конкатенира (събира) два стринга, връщайки като резултат нов стринг. Това позволява навързването на конкатениращи (+) операции една след друга във верига. Следващия пример показва конкатенация на три стринга.

```
string age = "twenty six";
string text = "He is " + age + " years old.";
Console.WriteLine(text);
```

Резултатът от изпълнението на този код е отново стринг:

```
He is twenty six years old.
```

Конкатенация на смесени типове

Какво се случва, когато искаме да отпечатаме по-голям и по-сложен текст, който се състои от различни типове? До сега използвахме версиите на метода **`WriteLine(…)`** за точно определен тип. Нужно ли е, когато искаме да отпечатаме различни типове наведнъж, да използваме различните версии на метода **`WriteLine(…)`** за всеки един от тези типове?

Отговорът на този въпрос е "не", тъй като в C# можем да съединяваме текстови и други данни (например числови) чрез оператора "**+**". Следващият пример е като предходния, но в него годините (**age**) са от целочислен тип, който е различен от стринг:

```
int age = 26;
string text = "He is " + age + " years old.";
Console.WriteLine(text);
```

В примера се извършва конкатенация и отпечатване. Резултатът от примера е следният:

```
He is 26 years old.
```

На втори ред от кода на примера виждаме, че се извършва операцията събиране (конкатенация) на стринга "**He is**" и целочисления тип "**age**". Опитваме се да съберем два различни типа. Това е възможно поради наличието на следващото важно правило.

Когато стринг участва в конкатенация с какъвто и да е друг тип, резултатът винаги е стринг.

От правилото става ясно, че резултатът от "**He is** " + **age** е отново стринг, след което резултатът се събира с последната част от израза " **years old.**". Така след извикване на верига от оператори за събиране, в крайна сметка се получава като резултат един стринг и съответно се извиква стринговата версия на метода **WriteLine(…)**.

За краткост, горният пример може да бъде написан и по следния начин:

```
int age = 26;
Console.WriteLine("He is " + age + " years old.");
```

Особености при конкатенация на низове

Има някои интересни ситуации при конкатенацията (съединяването) на низове, за които трябва да знаем и да внимаваме, защото водят до грешки. Следващият пример показва изненадващо поведение на код:

```
string s = "Four: " + 2 + 2;
Console.WriteLine(s);
// Four: 22

string s1 = "Four: " + (2 + 2);
Console.WriteLine(s1);
// Four: 4
```

Както се вижда от примера, редът на изпълнение на операторите (вж. главата "Оператори и изрази") е от голямо значение! В примера първо се извършва събиране на "**`Four: `**" с "`2`" и **резултатът от операцията е стринг**. Следва повторна конкатенация с второто число, от където се получава неочакваното слепване на резултата "**`Four: 22`**" вместо очакваното "**`Four: 4`**". Това е така, понеже операциите се изпълняват от ляво на дясно и винаги участва стринг в конкатенацията.

За да се избегне тази неприятна ситуация може да се използват скоби, които ще променят реда на изпълнение на операторите и ще се постигне желания резултат. Скобите, като оператори с най-голям приоритет, карат извършването на операцията "събиране" на двете числа да стане преди конкатенацията със стринг и така коректно се извършва първо събирането на двете числа, а след това съединяването със символния низ.

Посочената грешка е често срещана при начинаещи програмисти, защото те не съобразяват, че конкатенирането на символни низове се извършва отляво надясно, защото събирането на числа не е с по-висок приоритет, отколкото долепването на низове.

Когато конкатенирате низове и същевременно събирате числа, използвайте скоби, за да укажете правилния ред на операциите. Иначе те се изпълняват отляво надясно.

Форматиран изход с Write(…) и WriteLine(…)

За отпечатването на дълги и сложни поредици от елементи са въведени специални варианти (известни още като овърлоуди – overloads) на методите **`Write(…)`** и **`WriteLine(…)`**. Тези варианти имат съвсем различна концепция от тази на стандартните методи за печатане в C#. Основната им идея е да приемат специален стринг, форматиран със специални форматиращи символи и списък със стойностите, които трябва да се заместят на мястото на "форматните спецификатори". Ето как е дефиниран **`Write(…)`** в стандартните библиотеки на C#:

```
public static void Write(string format, object arg0,
   object arg1, object arg2, object arg3, … )
```

Форматиран изход – примери

Следващият пример отпечатва два пъти едно и също нещо, но по различен начин:

```
string str = "Hello World!";

// Print (the normal way)
Console.Write(str);

// Print (through formatting string)
Console.Write("{0}", str);
```

Резултатът от изпълнението на този пример е:

```
Hello World!Hello World!
```

Виждаме като резултат, два пъти "**`Hello, World!`**" на един ред. Това е така, понеже никъде в програмата не отпечатваме команда за нов ред.

Първо отпечатваме символния низ по познатия ни начин, за да видим разликата с другия подход. Второто отпечатване е форматиращото **`Write(…)`**, като първият аргумент е форматиращият стринг. В случая **`{0}`** означава, да се постави първият аргумент след форматиращия стринг **`str`** на мястото на **`{0}`**. Изразът **`{0}`** се нарича placeholder, т. е. място, което ще бъде заместен с конкретна стойност при отпечатването.

Следващият пример ще разясни допълнително концепцията:

```
string name = "Boris";
int age = 18;
string town = "Plovdiv";
Console.Write(
  "{0} is {1} years old from {2}!\n", name, age, town);
```

Резултатът от изпълнението на примера е следният:

```
Boris is 18 years old from Plovdiv!
```

От сигнатурата на тази версия на **`Write(…)`** видяхме че, първият аргумент е форматиращият низ. Следва поредица от аргументи, които се заместват на местата, където има цифра, оградена с къдрави скоби. Изразът **`{0}`** означава да се постави на негово място първият от аргументите, подаден след форматиращия низ, в случая **`name`**. Следва **`{1}`**, което означава, да се замести с втория от аргументите. Последният специален символ е **`{2}`**, което означава да се замести със следващия по ред параметър (**`town`**). Следва **`\n`**, което е специален символ, който указва минаване на нов ред.

Редно е да споменем, че всъщност командата за преминаване на нов ред под Windows е **`\r\n`**, а под Unix базирани операционни системи – **`\n`**. При работата с конзолата няма значение, че използваме само **`\n`**, защото стандартният входен поток възприема **`\n`** като **`\r\n`**, но ако пишем във файл, например, използването само на **`\n`** е грешно (под Windows).

Съставно форматиране

Методите за форматиран изход на класа **`Console`** използват така наречената **система за съставно форматиране** (composite formatting feature). Съставното форматиране се използва както при отпечатването на конзолата, така и при някои операции със стрингове. Вече разгледахме съставното форматиране в най-простия му вид в предишните примери, но то притежава много повече възможности от това, което видяхме. В основата си съставното форматиране използва две неща: **съставен форматиращ низ** и **поредица от аргументи**, които се заместват на определени места в низа.

Съставен форматиращ низ

Съставният форматиращ низ е смесица от нормален текст и **форматиращи елементи** (formatting items). При форматирането нормалният текст остава същият, както в низа, а на местата на форматиращите елементи се замества със стойностите на съответните аргументи, отпечатани според определени правила. Тези правила се задават чрез синтаксиса на форматиращите елементи.

Форматиращи елементи

Форматиращите елементи дават възможност за мощен контрол върху показваната стойност и затова могат да придобият доста сложен вид. Следващата схема на образуване показва общия синтаксис на **форматиращите елементи**:

```
{index[,alignment][:formatString]}
```

Както забелязваме, форматиращият елемент започва с отваряща къдрава скоба `{` и завършва със затваряща къдрава скоба `}`. Съдържанието между скобите е разделено на три компонента, като само **`index`** компонентата е задължителна. Сега ще разгледаме всяка една от тях поотделно.

Index компонента

`Index` компонентата е цяло число и показва позицията на аргумента от списъка с аргументи. Първият аргумент се обозначава с "`0`", вторият с "`1`" и т.н. В съставния форматиращ низ е позволено да има множество форматиращи елементи, които се отнасят за един и същ аргумент. В този случай **`index`** компонентата на тези елементи е едно и също число. Няма ограничение за последователността на извикване на аргументите. Например бихме могли да използваме следния форматиращ низ:

```
Console.Write(
  "{1} is {0} years old from {2}!", 18, "Peter", "Plovdiv");
```

В случаите, когато някой от аргументите не е рефериран от никой от форматиращите елементи, той просто се пренебрегва и не играе никаква роля. Въпреки това е добре такива аргументи да се премахват от списъка с аргументи, защото внасят излишна сложност и могат да доведат до объркване. В обратния случай – когато форматиращ елемент реферира аргумент, който не съществува в списъка от аргументи, се хвърля изключение. Това може да се получи, например, ако имаме форматиращ елемент `{4}`, а сме подали списък със само два аргумента.

Alignment компонента

`Alignment` компонентата е незадължителна и указва подравняване на стринга. Тя е цяло положително или отрицателно число, като положителните стойности означават подравняване от дясно, а отрицателните – от ляво. Стойността на числото обозначава броя на позициите, в които да се подравни числото. Ако стрингът, който искаме да изобразим има дължина по-голяма или равна на стойността на числото, тогава това число се пренебрегва. Ако е по-малка обаче, незаетите позиции се допълват с интервали. Например следното форматиране:

```
Console.WriteLine("{0,6}", 123);
Console.WriteLine("{0,6}", 1234);
Console.WriteLine("{0,6}", 12);
```

ще изведе следния резултат:

```
   123
  1234
    12
```

Ако решим да използваме alignment компонента, тя трябва да е отделена от **`index`** компонентата чрез запетая, както е направено в примера по-горе.

FormatString компонента

Тази компонента указва специфичното форматиране на низа. Тя варира в зависимост от типа на аргумента. Различават се три основни типа **`formatString`** компоненти:

- за числени типове аргументи
- за аргументи от тип дата (**`DateTime`**)
- за аргументи от тип енумерация (изброени типове)

FormatString компоненти за числа

Този тип **`formatString`** компонента има два подтипа: стандартно дефинирани формати и формати дефинирани от потребителя (custom format strings).

Стандартно дефинирани формати за числа

Тези формати се дефинират чрез един от няколко **форматни спецификатора**, които представляват буква със специфично значение. След форматния спецификатор може да следва цяло положително число, наречено **прецизност**, което за различните спецификатори има различно значение. Когато тя има значение на брой знаци след десетичната запетая, тогава резултатът се закръгля. Следната таблица описва спецификаторите и значението на прецизността:

Спецификатор	Описание
`"C"` **или** `"c"`	Обозначава валута и резултатът ще се изведе заедно със знака на валутата за текущата "култура" (например българската). Прецизността указва броя на знаците след десетичната запетая.
`"D"` **или** `"d"`	Цяло число. Прецизността указва минималния брой знаци за изобразяването на стринга, като при нужда се извършва допълване с нули отпред.
`"E"` **или** `"e"`	Експоненциален запис. Прецизността указва броя на знаците след десетичната запетая.
`"F"` **или** `"f"`	Цяло или дробно число. Прецизността указва броя на знаците след десетичната запетая.
`"N"` **или** `"n"`	Еквивалентно на "F", но изобразява и съответния разделител за хилядите, милионите и т.н. (например в английския език често числото "1000" се изписва като "1,000" - със запетая между числото 1 и нулите).
`"P"` **или** `"p"`	Ще умножи числото по 100 и ще изобрази отпред символа за процент. Прецизността указва броя на знаците след десетичната запетая.
`"X"` **или** `"x"`	Изписва числото в шестнадесетична бройна система. Работи само с цели числа. Прецизността указва минималния брой знаци за изобразяването на стринга, като недостигащите се допълват с нули отпред.

Част от форматирането се определя от текущите настройки за "култура", които се взимат по подразбиране от регионалните настройки на операционната система. "Културите" са набор от правила, които са валидни за даден език или за дадена държава и които указват, кой символ да се използва за десетичен разделител, как се изписва валутата и др. Например, за българската "култура" валутата се изписва като след сумата се добавя " лв.", докато за американската "култура" се изписва символът "$" преди сумата. Нека видим и няколко примера за използването на спецификаторите от горната таблица при регионални настройки за български език:

StandardNumericFormats.cs

```
class StandardNumericFormats
{
   static void Main()
   {
      Console.WriteLine("{0:C2}", 123.456);
      //Output: 123,46 лв.
      Console.WriteLine("{0:D6}", -1234);
      //Output: -001234
      Console.WriteLine("{0:E2}", 123);
      //Output: 1,23E+002
      Console.WriteLine("{0:F2}", -123.456);
      //Output: -123,46
      Console.WriteLine("{0:N2}", 1234567.8);
      //Output: 1 234 567,80
      Console.WriteLine("{0:P}", 0.456);
      //Output: 45,60 %
      Console.WriteLine("{0:X}", 254);
      //Output: FE
    }
}
```

Потребителски формати за числа

Всички формати, които не са стандартни, се причисляват към потребителските (custom) формати. За custom форматите отново са дефинирани набор от спецификатори, като разликата със стандартните формати е, че може да се използват поредица от спецификатори (при стандартните формати се използва само един спецификатор от възможните). В следващата таблица са изброени различните спецификатори и тяхното значение:

Спецификатор	Описание
0	Обозначава цифра. Ако на тази позиция в резултата липсва цифра, се изписва цифрата 0.
#	Обозначава цифра. Не отпечатва нищо, ако на тази позиция в резултата липсва цифра.
.	Десетичен разделител за съответната "култура".
,	Разделител за хилядите в съответната "култура".
%	Умножава резултата по 100 и отпечатва символ за процент.
E0 или E+0 или E-0	Обозначава експоненциален запис. Броят на нулите указва броя на знаците на експонентата. Знакът "+" обозначава, че искаме винаги да изпишем и знакът на числото, докато минус означава да се изпише знакът, само ако стойността е отрицателна.

При използването на custom формати за числа има доста особености, но те няма да се обсъждат тук, защото темата ще се измести в посока, в която няма нужда. Ето няколко по-прости примера, които илюстрират как се използват потребителски форматиращи низове:

CustomNumericFormats.cs

```
class CustomNumericFormats
{
   static void Main()
   {
      Console.WriteLine("{0:0.00}", 1);
      //Output: 1,00
      Console.WriteLine("{0:#.##}", 0.234);
      //Output: ,23
      Console.WriteLine("{0:#####}", 12345.67);
      //Output: 12346
      Console.WriteLine("{0:(0#) ### ## ##}", 29342525);
      //Output: (02) 934 25 25
      Console.WriteLine("{0:%##}", 0.234);
      //Output: %23
   }
}
```

FormatString компоненти за дати

При форматирането на дати отново имаме разделение на стандартни и custom формати за дати.

Стандартно дефинирани формати за дати

Тъй като стандартно дефинираните формати са доста, ще изброим само някои от тях. Останалите могат лесно да бъдат проверени в MSDN.

Спецификатор	Формат (за българска "култура")
`d`	23/10/2009 г.
`D`	23 Октомври 2009 г.
`t`	15:30 (час)
`T`	15:30:22 ч. (час)
`Y` **или** `y`	Октомври 2009 г. (само месец и година)

Custom формати за дати

Подобно на custom форматите за числа и тук имаме множество форматни спецификатори, като можем да комбинираме няколко от тях. Тъй като и тук тези спецификатори са много, ще покажем само някои от тях, с които да демонстрираме как се използват custom форматите за дати. Разгледайте следната таблица:

Спецификатор	Формат (за българска "култура")
`d`	Ден – от 0 до 31
`dd`	Ден – от 00 до 31
`M`	Месец – от 0 до 12
`MM`	Месец – от 00 до 12

yy	Последните две цифри на годината (от 00 до 99)
yyyy	Година, изписана с 4 цифри (например 2011)
hh	Час – от 00 до 11
HH	Час – от 00 до 23
m	Минути – от 0 до 59
mm	Минути – от 00 до 59
s	Секунди – от 0 до 59
ss	Секунди – от 00 до 59

При използването на тези спецификатори можем да вмъкваме различни разделители между отделните части на датата, като например "." или "/". Ето няколко примера:

```
DateTime d = new DateTime(2009, 10, 23, 15, 30, 22);
Console.WriteLine("{0:dd/MM/yyyy HH:mm:ss}", d);
Console.WriteLine("{0:d.MM.yy г.}", d);
```

При изпълнение на примерите се получава следният резултат:

```
23.10.2009 15:30:22
23.10.09 г.
```

FormatString компоненти за енумерации

Енумерациите (изброени типове) представляват типове данни, които могат да приемат като стойност една измежду няколко предварително дефинирани възможни стойности (например седемте дни от седмицата). Ще ги разгледаме подробно в темата "Дефиниране на класове".

При енумерациите почти няма какво да се форматира. Дефинирани са четири стандартни форматни спецификатора:

Спецификатор	Формат (за българска "култура")
G или g	Представя енумерацията като стринг.
D или d	Представя енумерацията като число.
X или x	Представя енумерацията като число в шестнадесетичната бройна система и с осем цифри.

Ето няколко примера:

```
Console.WriteLine("{0:G}", DayOfWeek.Wednesday);
Console.WriteLine("{0:D}", DayOfWeek.Wednesday);
Console.WriteLine("{0:X}", DayOfWeek.Wednesday);
```

При изпълнение на горния код получаваме следния резултат:

```
Wednesday
3
00000003
```

Форматиращи низове и локализация

При използването на форматиращи низове е възможно една и съща програма да отпечатва различни стойности в зависимост от настройките за локализация в операционната система. Например, при отпечатване на месеца от дадена дата, ако текущата локализация е българската, ще се отпечата на български, примерно "Август", докато ако локализацията е американската, ще се отпечата на английски, примерно "August".

При стартирането на конзолното приложение, то автоматично извлича системната локализация на операционната система и ползва нея за четене и писане на форматирани данни (числа, дати и други).

Локализацията в .NET се нарича още "култура" и може да се променя ръчно чрез класа **`System.Globalization.CultureInfo`**. Ето един пример, в който отпечатваме едно число и една дата по американската и по българската локализация:

CultureInfoExample.cs

```
using System;
using System.Threading;
using System.Globalization;

class CultureInfoExample
{
   static void Main()
   {
      DateTime d = new DateTime(2009, 10, 23, 15, 30, 22);

      Thread.CurrentThread.CurrentCulture =
         CultureInfo.GetCultureInfo("en-US");
      Console.WriteLine("{0:N}", 1234.56);
      Console.WriteLine("{0:D}", d);

      Thread.CurrentThread.CurrentCulture =
         CultureInfo.GetCultureInfo("bg-BG");
      Console.WriteLine("{0:N}", 1234.56);
      Console.WriteLine("{0:D}", d);
   }
}
```

При стартиране на примера се получава следният резултат:

```
1,234.56
Friday, October 23, 2009
```

```
1 234,56
23 Октомври 2009 г.
```

Вход от конзолата

Както в началото на темата обяснихме, най-подходяща за малки приложения е конзолната комуникация, понеже е най-лесна за имплементиране. **Стандартното входно устройство** е тази част от операционната система, която контролира от къде програмата ще получи своите входни данни. По подразбиране "стандартното входно устройство" чете своя вход от драйвер "закачен" за клавиатурата. Това може да бъде променено и стандартният вход може да бъде пренасочен към друго място, например към файл, но това се прави рядко.

Всеки език за програмиране има механизъм за четене и писане в конзолата. Обектът, контролиращ стандартния входен поток в **C#**, е **`Console.In`**.

От конзолата можем да четем различни данни:

- текст;
- други типове, след "парсване" на текста.

Реално за четене рядко се използва стандартният входен поток **`Console.In`** директно. Класът **`Console`** предоставя два метода **`Console.Read()`** и **`Console.ReadLine()`**, които работят върху този поток и обикновено четенето от конзолата се осъществява чрез тях.

Четене чрез Console.ReadLine()

Най-голямо удобство при четене от конзолата предоставя методът **`Console.ReadLine()`**. Как работи той? При извикването му програмата преустановява работата си и чака за вход от конзолата. Потребителят въвежда някакъв стринг в конзолата и натиска клавишът [Enter]. В този момент конзолата разбира, че потребителят е свършил с въвеждането и прочита стринга. Методът **`Console.ReadLine()`** връща като резултат въведения от потребителя стринг. Сега може би е ясно, защо този метод има такова име.

Следващият пример демонстрира работата на **`Console.ReadLine()`**:

UsingReadLine.cs

```
class UsingReadLine
{
  static void Main()
  {
    Console.Write("Please enter your first name: ");
    string firstName = Console.ReadLine();

    Console.Write("Please enter your last name: ");
    string lastName = Console.ReadLine();

    Console.WriteLine("Hello, {0} {1}!", firstName, lastName);
  }
}

// Output:  Please enter your first name: Iliyan
//          Please enter your last name: Murdanliev
//          Hello, Iliyan Murdanliev!
```

Виждаме колко лесно става четенето на текст от конзолата с метода **`Console.ReadLine()`**:

- Отпечатваме текст в конзолата, който пита за името на потребителя.
- Извършваме четене на цял ред от конзолата, чрез метода **`ReadLine()`**. Това води до блокиране на програмата докато потребителят не въведе някакъв текст и не натисне [Enter].
- Повтаряме горните две стъпки и за фамилията.
- След като сме събрали необходимата информация я отпечатваме в конзолата.

Четене чрез Console.Read()

Методът **`Read()`** работи по малко по-различен начин от **`ReadLine()`**. Като за начало той прочита само един символ, а не цял ред. Другата основна разлика е че методът не връща директно прочетения символ, а само неговия код. Ако желаем да използваме резултата като символ, трябва да го преобразуваме към символ или да използваме метода **`Convert.ToChar()`** върху него. Има и една важна особеност: символът се прочита чак когато се натисне клавишът [Enter]. Тогава целият стринг написан на конзолата се прехвърля в буфера на стандартния входен поток и методът **`Read()`** прочита първия символ от него. При последващи извиквания на метода, ако буферът не е празен (т.е. има вече въведени, но все още непрочетени символи), то изпълнението на програмата няма да спре и да чака, а директно ще прочете следващия символ от буфера и така докато буферът не се изпразни. Едва тогава програмата ще чака наново за потребителски вход, ако отново се извика **`Read()`**. Ето един пример:

UsingRead.cs

```
class UsingRead
{
  static void Main()
  {
    int codeRead = 0;
```

```
      do
      {
        codeRead = Console.Read();
        if (codeRead != 0)
        {
          Console.Write((char)codeRead);
        }
      }
      while (codeRead != 10);
    }
}
```

Тази програма чете един ред от потребителя и го отпечатва символ по символ. Това става възможно благодарение на малка хитринка - предварително знаем, че клавишът Enter всъщност вписва в буфера два символа. Това са "carriage return" код (`ASCII 13`) следван от "linefeed" код (`ASCII 10`). За да разберем, че един ред е свършил ние търсим за символ с код 10. По този начин програмата прочита само един ред и излиза от цикъла.

Трябва да споменем, че методът `Console.Read()` почти не се използва в практиката, при наличието на алтернативата с `Console.ReadLine()`. Причината за това е, че вероятността да сгрешим с `Console.Read()` е доста по-голяма отколкото ако изберем алтернативен подход, а кодът най-вероятно ще е ненужно сложен.

Четене на числа

Четенето на числа от конзолата в C# не става директно. За да прочетем едно число, преди това трябва да прочетем входа като стринг (чрез `ReadLine()`) и след това да преобразуваме този стринг в число. Операцията по преобразуване от стринг в някакъв друг тип се нарича **парсване**. Всички примитивни типове имат методи за парсване. Ще дадем един прост пример за четене и парсване на числа:

ReadingNumbers.cs

```
class ReadingNumbers
{
  static void Main()
  {
    Console.Write("a = ");
    int a = int.Parse(Console.ReadLine());

    Console.Write("b = ");
    int b = int.Parse(Console.ReadLine());

    Console.WriteLine("{0} + {1} = {2}", a, b, a + b);
    Console.WriteLine("{0} * {1} = {2}", a, b, a * b);
    Console.Write("f = ");
    double f = double.Parse(Console.ReadLine());
    Console.WriteLine("{0} * {1} / {2} = {3}",
      a, b, f, a * b / f);
  }
}
```

Резултатът от изпълнението на програмата би могъл да е следният (при условие че въведем 5, 6 и 7.5 като входни данни):

```
a = 5
b = 6
5 + 6 = 11
5 * 6 = 30
f = 7,5
5 * 6 / 7,5 = 4
```

В този пример особеното е, че използваме методите за парсване на числени типове и при грешно подаден резултат (например текст), ще възникне грешка (изключение) **`System.FormatException`**. Това важи с особена сила при четенето на реално число, защото разделителят, който се използва между цялата и дробната част, е различен при различните култури и зависи от регионалните настройки на операционната система.

Разделителят за числата с плаваща запетая зависи от текущите езикови настройки на операционната система (Regional and Language Options в Windows). При едни системи за разделител може да се счита символът запетая, при други точка. Въвеждането на точка вместо запетая ще предизвика `System.FormatException`.

Изключенията като механизъм за съобщаване на грешки ще разгледаме в главата "Обработка на изключения". За момента можете да считате, че когато програмата даде грешка, това е свързано с възникването на изключение, което отпечатва детайлна информация за грешката на конзолата. За пример нека предположим, че регионалните настройки на компютъра са българските и че изпълняваме следния код:

```
Console.Write("Enter a floating-point number: ");
string line = Console.ReadLine();
double number = double.Parse(line);
Console.WriteLine("You entered: {0}", number);
```

Ако въведем числото "3.14" (с грешен за българските настройки разделител "."), ще получим следното изключение (съобщение за грешка):

```
Unhandled Exception: System.FormatException: Input string was
```

```
not in a correct format.
   at System.Number.StringToNumber(String str, NumberStyles
options, NumberBuffer& number, NumberFormatInfo info, Boolean
parseDecimal)
   at System.Number.ParseDouble(String value, NumberStyles
options, NumberFormatInfo numfmt)
   at System.Double.Parse(String s, NumberStyles style,
NumberFormatInfo info)
   at System.Double.Parse(String s)
   at ConsoleApplication.Program.Main() in
C:\Projects\IntroCSharpBook\ConsoleExample\Program.cs:line 14
```

Условно парсване на числа

При парсване на символен низ към число чрез метода **`Int32.Parse(string)`** или чрез **`Convert.ToInt32(string)`** ако подаденият символен низ не е число, се получава изключение. Понякога се налага да се прихване неуспешното парсване и да се отпечата съобщение за грешка или да се помоли потребителя да въведе нова стойност.

Прихващането на грешно въведено число при парсване на символен низ може да стане по два начина:

- чрез прихващане на изключения (вж. главата "Обработка на изключения");
- чрез условно парсване (посредством метода **`TryParse(…)`**).

Нека разгледаме условното парсване на числа в .NET Framework. Методът **`Int32.TryParse(…)`** приема два параметъра – стринг за парсване и променлива за записване на резултата от парсването. Ако парсването е успешно, методът връща стойност е **`true`**. За повече яснота, нека разгледаме един пример:

```
string str = Console.ReadLine();
int intValue;
bool parseSuccess = Int32.TryParse(str, out intValue);
Console.WriteLine(parseSuccess ?
   "The square of the number is " + intValue * intValue + "."
   : "Invalid number!");
```

В примера се извършва условно парсване на стринг въведен от конзолата към целочисления тип **`Int32`**. Ако въведем като вход "**2**", тъй като парсването ще бъде успешно, резултатът от **`TryParse()`** ще бъде **`true`**, в променливата **`intValue`** ще бъде записано парснатото число и на конзолата ще се отпечата въведеното число на квадрат:

```
Result: The square of the number is 4.
```

Ако опитаме да парснем невалидно число, например "**`abc`**", **`TryParse()`** ще върне резултат **`false`** и на потребителя ще бъде обяснено, че е въвел невалидно число:

```
Invalid number!
```

Обърнете внимание, че методът **`TryParse()`** в резултат на своята работа връща едновременно две стойности: парсనатото число (като изходен параметър) и булева стойност като резултат от извикването на метода. Връщането на няколко стойности едновременно е възможно, тъй като едната стойност се връща като **изходен параметър** (**`out`** параметър). Изходните параметри връщат стойност в предварително зададена за целта променлива съвпадаща с техния тип. При извикване на метод изходните параметри се предшестват задължително от ключовата дума **`out`**.

Четене чрез Console.ReadKey()

Методът **`Console.ReadKey()`** изчаква натискане на клавиш на конзолата и прочита неговият символен еквивалент, без да е необходимо да се натиска [Enter]. Резултатът от извикването на **`ReadKey()`** е информация за натиснатия клавиш (или по-точно клавишна комбинация), във вид на обект от тип **`ConsoleKeyInfo`**. Полученият обект съдържа символа, който се въвежда чрез натиснатата комбинация от клавиши (свойство **`KeyChar`**), заедно с информация за клавишите [Shift], [Ctrl] и [Alt] (свойство **`Modifiers`**). Например, ако натиснем [Shift+A], ще прочетем главна буква '**`A`**', а в свойството Modifiers ще присъства флага **`Shift`**. Следва пример:

```
ConsoleKeyInfo key = Console.ReadKey();
Console.WriteLine();
Console.WriteLine("Character entered: " + key.KeyChar);
Console.WriteLine("Special keys: " +  key.Modifiers);
```

Ако изпълним програмата и натиснем [Shift+A], ще получим следния резултат:

```
A
Character entered: A
Special keys: Shift
```

Вход и изход на конзолата – примери

Ще разгледаме още няколко примера за вход и изход от конзолата, с които ще ви покажем още няколко интересни техники.

Печатане на писмо

Следва един практичен пример, показващ конзолен вход и форматиран текст под формата на писмо.

PrintingLetter.cs

```
class PrintingLetter
{
  static void Main()
  {
    Console.Write("Enter person name: ");
    string person = Console.ReadLine();

    Console.Write("Enter book name: ");
    string book = Console.ReadLine();

    string from = "Authors Team";

    Console.WriteLine("  Dear {0},", person);
    Console.Write("We are pleased to inform " +
      "you that \"{1}\" is the best Bulgarian book. {2}" +
      "The authors of the book wish you good luck {0}!{2}",
      person, book, Environment.NewLine);

    Console.WriteLine("  Yours,");
    Console.WriteLine("  {0}", from);
  }
}
```

Резултатът от изпълнението на горната програма би могъл да е следния:

```
Enter person name: Readers
Enter book name: Introduction to programming with C#
  Dear Readers,
We are pleased to inform you that "Introduction to programming
with C#" is the best Bulgarian book.
The authors of the book wish you good luck Readers!
  Yours,
  Authors Team
```

В този пример имаме предварителен шаблон на писмо. Програмата "задава" няколко въпроса на потребителя и прочита от конзолата нужната информация, за да отпечата писмото, като замества форматиращите спецификатори с попълнените от потребителя данни.

Лице на правоъгълник или триъгълник

Ще разгледаме още един пример: изчисляване на лице на правоъгълник или триъгълник.

CalculatingArea.cs

```
class CalculatingArea
```

```
{
  static void Main()
  {
    Console.WriteLine("This program calculates " +
    "the area of a rectangle or a triangle");

    Console.WriteLine("Enter a and b (for rectangle) " +
      "or a and h (for triangle): ");

    int a = int.Parse(Console.ReadLine());
    int b = int.Parse(Console.ReadLine());

    Console.WriteLine("Enter 1 for a rectangle or " +
         "2 for a triangle: ");

    int choice = int.Parse(Console.ReadLine());
    double area = (double) (a * b) / choice;
    Console.WriteLine("The area of your figure is " + area);
  }
}
```

Резултатът от изпълнението на горния пример е следният:

```
This program calculates the area of a rectangle or a triangle
Enter a and b (for rectangle) or a and h (for triangle):
5
4
Enter 1 for a rectangle or 2 for a triangle:
2
The area of your figure is 10
```

Упражнения

1. Напишете програма, която чете от конзолата три числа от тип **`int`** и отпечатва тяхната сума.

2. Напишете програма, която чете от конзолата радиуса "**`r`**" на кръг и отпечатва неговия периметър и обиколка.

3. Дадена фирма има име, адрес, телефонен номер, факс номер, уеб сайт и мениджър. Мениджърът има име, фамилия и телефонен номер. Напишете програма, която чете информацията за фирмата и нейния мениджър и я отпечатва след това на конзолата.

4. Напишете програма, която отпечатва три числа в три виртуални колони на конзолата. Всяка колона трябва да е с широчина 10 символа, а числата трябва да са ляво подравнени. Първото число трябва да е цяло число в шестнадесетична бройна система, второто да е дробно положително, а третото – да е дробно отрицателно.

Последните две числа да се закръглят до втория знак след десетичната запетая.

5. Напишете програма, която чете от конзолата две цели числа (**`int`**) и отпечатва, колко числа има между тях, такива, че остатъкът им от делението на 5 да е 0. Пример: в интервала (17, 25) има 2 такива числа.
6. Напишете програма, която чете две числа от конзолата и отпечатва по-голямото от тях. Решете задачата без да използвате условни конструкции.
7. Напишете програма, която чете пет числа и отпечатва тяхната сума. При невалидно въведено число да се подкани потребителя да въведе друго число.
8. Напишете програма, която чете пет числа от конзолата и отпечатва най-голямото от тях.
9. Напишете програма, която чете коефициентите **`a`**, **`b`** и **`c`** от конзолата и решава уравнението: **$ax^2+bx+c=0$**. Програмата трябва да принтира реалните решения на уравнението на конзолата.
10. Напишете програма, която прочита едно цяло число **`n`** от конзолата. След това прочита още **`n`** на брой числа от конзолата и отпечатва тяхната сума.
11. Напишете програма, която прочита цяло число **`n`** от конзолата и отпечатва на конзолата всички числа в интервала **`[1…n]`**, всяко на отделен ред.
12. Напишете програма, която отпечатва на конзолата първите 100 числа от редицата на Фибоначи: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...
13. Напишете програма, която пресмята сумата (с точност до 0.001): 1+ 1/2 + 1/3 + 1/4 + 1/5 + ...

Решения и упътвания

1. Използвайте методите **`Console.ReadLine()`** и **`Int32.Parse()`**.
2. Използвайте константата **`Math.PI`** и добре известните формули от планиметрията.
3. Форматирайте текста с **`Write(…)`** или **`WriteLine(…)`** подобно на този от примера с писмото, който разгледахме.
4. Използвайте форматиращите настройки, предоставени от съставното форматиране и метода **`Console.WriteLine()`**.
5. Има два подхода за решаване на задачата:

Първи подход: Използват се математически хитрини за оптимизирано изчисляване, базирани на факта, че всяко пето число се дели на 5.

Вторият подход е по-лесен, но работи по-бавно. Чрез **for** цикъл може да се провери всяко число в дадения интервал. За целта трябва да прочетете от Интернет или от главата "Цикли" как се използва **for** цикъл.

6. Тъй като в задачата се иска решение, което не използва условни оператори, трябва да подходите по по-различен начин. Две от възможните решения на задачата включват използване на функции от класа **Math**:

   - По-голямото от двете числа можете да намерите с функцията **Math.Max(a, b)**, а по-малкото с **Math.Min(a, b)**.
   - Друго решение на задачата включва използването на функцията за взимане на абсолютна стойност на число **Math.Abs(a)**:

```
int a = 2011;
int b = 1990;
Console.WriteLine("Greater: {0}", (a + b + Math.Abs(a - b)) / 2);
Console.WriteLine("Smaller: {0}", (a + b - Math.Abs(a - b)) / 2);
```

   Третото решение използва побитови операции:

```
int a = 1990;
int b = 2011;
int max = a - ((a - b) & ((a - b) >> 31));
Console.WriteLine(max);
```

7. Можете да прочетете числата в пет различни променливи и накрая да ги сумирате. При парсване на поредното число използвайте условно парсване с **TryParse(…)**. При въведено невалидно число повторете четенето на число. Можете да сторите това чрез **while** цикъл с подходящо условие за изход. За да няма повторение на код, можете да разгледате конструкцията за цикъл "**for**" от главата "Цикли".

8. Трябва да използвате конструкцията за сравнение "**if**", за която можете да прочетете в Интернет или от главата "Условни конструкции". За да избегнете повторението на код, можете да използвате конструкцията за цикъл "**for**", за която също трябва да прочетете в Интернет или от главата "Цикли".

9. Използвайте добре познатия метод за решаване на квадратни уравнения. Разгледайте внимателно всички възможни случаи.

10. Четете числата едно след друго и натрупвайте тяхната сума в променлива, която накрая изведете на конзолата.

11. Използвайте комбинация от цикли и методите **Console.ReadLine()**, **Console.WriteLine()** и **Int32.Parse()**.

12. Повече за редицата на Фибоначи можете да намерите в Wikipedia на адрес: http://en.wikipedia.org/wiki/Fibonacci_sequence. За решение на задачата използвайте 2 временни променливи, в които да пазите последните 2 пресметнати стойности и с цикъл пресмятайте останалите (всяко следващо число в редицата е сума от последните две).

13. Натрупвайте сумата в променлива с цикъл и пазете старата сума, докато разликата между двете суми стане по-малка от точността (0.001).

Telerik Academy

SCHOOL

Стани част от Академията на Телерик за ученици!

“Основната цел на обучението е да израстнеш, а нашите умове, за разлика от телата ни, могат да растат през целия ни живот!“ Мортимър Адлър

АКАДЕМИЯТА НА ТЕЛЕРИК за ученици предоставя съвременно, специализирано обучение по софтуерно инженерство за ученици от средните училища в България. Нашата основна цел е да помогнем на младите момчета и момичета да се подготвят за Националната олимпиада по информационни технологии (НОИТ) и да придобият солидни теоретични и практически умения за разработка на софтуер и софтуерни технологии.

В Училищната академия на Телерик по софтуерно инженерство:

- **Стимулираме креативността и мисленето на учениците**
- **Всеки ученик получава индивидуално внимание и съдействие**
- **Полагаме основите за писане на качествен програмен код**
- **Изучаваме съвременните софтуерни технологии**
- **Подготвяме учениците за Националната олимпиада по информационни технологии**
- **Предлагаме голямо разнообразие от практически и теоретични обучения**

Програмата на Академията включва:

C#, .NET технологии, бази данни, SQL, Silverlight, 3D графики и моделиране, разработка на игри, програмиране на микроконтролери, WPF, ASP.NET, HTML5, разработка на мобилни приложения, основи на софтуерното инженерство, работа в екип

schoolacademy.telerik.com

groups.google.com/group/it-olymp *facebook.com/TelerikSchoolAcademy*

telerik

deliver more than expected

Глава 5. Условни конструкции

В тази тема...

В настоящата тема ще разгледаме условните конструкции в езика C#, чрез които можем да изпълняваме различни действия в зависимост от някакво условие. Ще обясним синтаксиса на условните оператори: **`if`** и **`if-else`** с подходящи примери и ще разясним практическото приложение на оператора за избор **`switch`**.

Ще обърнем внимание на добрите практики, които е нужно да бъдат следвани, с цел постигане на по-добър стил на програмиране при използването на вложени или други видове условни конструкции.

Оператори за сравнение и булеви изрази

В следващата секция ще припомним основните оператори за сравнение в езика C#. Те са важни, тъй като чрез тях описваме условия при използването на условни конструкции.

Оператори за сравнение

В C# има няколко оператора за сравнение, които се използват за сравняване на двойки цели числа, числа с плаваща запетая, символи, низове и други типове данни:

Оператор	Действие
`==`	равно
`!=`	различно
>	по-голямо
>=	по-голямо или равно
<	по-малко
<=	по-малко или равно

Операторите за сравнение могат да сравняват произволни изрази, например две числа, два числови израза или число и променлива. Резултатът от сравнението е булева стойност (**`true`** или **`false`**).

Нека погледнем един пример, в който използваме сравнения:

```
int weight = 700;
Console.WriteLine(weight >= 500); // True

char gender = 'm';
Console.WriteLine(gender <= 'f'); // False

double colorWaveLength = 1.630;
Console.WriteLine(colorWaveLength > 1.621); // True

int a = 5;
int b = 7;
bool condition = (b > a) && (a + b < a * b);
Console.WriteLine(condition); // True

Console.WriteLine('B' == 'A' + 1); // True
```

В примерния програмен код извършваме сравнение между числа и между символи. При сравнението на числа те се сравняват по големина, а при сравнението на символи се сравнява тяхната лексикографска подредба (сравняват се Unicode номерата на съответните символи).

Както се вижда от примера, типът **`char`** има поведение на число и може да бъде събиран, изваждан и сравняван свободно с числа, но тази възможност трябва да се ползва внимателно, защото може да доведе до труден за четене и разбиране код.

Стартирайки примера ще получим следния резултат:

```
True
False
True
True
True
```

В C# има няколко различни типа данни, които могат да бъдат сравнявани:

- числа (**`int`**, **`long`**, **`float`**, **`double`**, **`ushort`**, **`decimal`**, …)
- символи (**`char`**)
- булеви стойности (**`bool`**)
- референции към обекти, познати още като обектни указатели (**`string`**, **`object`**, масиви и други)

Всяко едно сравнение може да засегне две числа, две **`bool`** стойности, или две референции към обекти. Позволено е да се сравняват изрази от различни типове, например цяло число с число с плаваща запетая, но не всяка двойки типове данни могат директно да се сравняват. Например не можем да сравняваме стринг с число.

Сравнение на цели числа и символи

Когато се сравняват числа и символи, се извършва сравнение директно между техните двоични представяния в паметта, т. е. сравняват се техните стойности. Например, ако сравняваме две числа от тип **`int`**, ще бъдат сравнени стойностите на съответните поредици от 4 байта, които ги съставят. Ето един пример за сравнение на символи и числа:

```
Console.WriteLine("char 'a' == 'a'? " + ('a' == 'a'));  // True
Console.WriteLine("char 'a' == 'b'? " + ('a' == 'b'));  // False
Console.WriteLine("5 != 6? " + (5 != 6));               // True
Console.WriteLine("5.0 == 5L? " + (5.0 == 5L));         // True
Console.WriteLine("true == false? " + (true == false)); // False
```

Резултатът от примера изглежда по следния начин:

```
char 'a' == 'a'? True
char 'a' == 'b'? False
5 != 6? True
5.0 == 5L? True
true == false? False
```

Сравнение на референции към обекти

В .NET Framework съществуват референтни типове данни, които не съдържат директно стойността си (както числовите типове), а съдържат адрес от динамичната памет, където е записана стойността им. Такива типове са стринговете, масивите и класовете. Те имат поведение на указател към някакви стойности и могат да имат стойност **`null`**, т.е. липса на стойност. При сравняването на променливи от референтен тип се сравняват адресите, които те пазят, т.е. проверява се дали сочат на едно и също място в паметта, т.е. към един и същ обект.

Два указателя към обекти (референции) могат да сочат към един и същи обект или към различни обекти или някой от тях може да не сочи никъде (да има стойност **`null`**). В следващия пример създаваме две променливи, които сочат към една и съща стойност (обект) в динамичната памет:

```
string str = "beer";
string anotherStr = str;
```

След изпълнението на този код, двете променливи **`str`** и **`anotherStr`** ще сочат към един и същи обект (**`string`** със стойност "beer"), който се намира на някакъв адрес в динамичната памет (managed heap).

Променливите от тип референция към обект могат да бъдат проверени, дали сочат към един и същ обект, посредством оператора за сравнение **==**. За повечето референтни типове този оператор не сравнява съдържанието на обектите, а само дали се намират на едно и също място в паметта, т. е. дали са един и същ обект. За променливи от тип обект, не са приложими сравненията по големина (**<**, **>**, **<=** и **>=**).

Следващият пример илюстрира сравнението на референции към обекти:

```
string str = "beer";
string anotherStr = str;
string thirdStr = "be" + 'e' + 'r';
Console.WriteLine("str = {0}", str);
Console.WriteLine("anotherStr = {0}", anotherStr);
Console.WriteLine("thirdStr = {0}", thirdStr);
Console.WriteLine(str == anotherStr); // True - same object
Console.WriteLine(str == thirdStr); // True - equal objects
Console.WriteLine((object)str == (object)anotherStr); // True
Console.WriteLine((object)str == (object)thirdStr); // False
```

Ако изпълним примера, ще получим следния резултат:

```
str = beer
anotherStr = beer
thirdStr = beer
True
True
```

```
True
False
```

Понеже стринговете, използвани в примера (инстанциите на класа **`System.String`**, дефинирани чрез ключовата дума **`string`** в C#), са от референтен тип, техните стойности се заделят като обекти в динамичната памет. Двата обекта, които се създават **`str`** и **`thirdStr`** имат равни стойности, но са различни обекти, разположени на различни адреси в паметта. Променливата **`anotherStr`** също е от референтен тип и приема адреса (референцията) на **`str`**, т.е. сочи към вече съществуващия обект **`str`**. Така при сравнението на променливите **`str`** и **`anotherStr`** се оказва, че те са един и същ обект и съответно са равни. При сравнението на **`str`** с **`thirdStr`** резултатът също е равенство, тъй като операторът **`==`** сравнява стринговете по стойност, а не по адрес (едно много полезно изключение от правилото за сравнение по адрес). Ако обаче преобразуваме трите променливи към обекти и тогава ги сравним, ще получим сравнение на адресите, където стоят стойностите им в паметта и резултатът ще е различен. Това показва, че операторът **`==`** има специално поведение, когато се сравняват стрингове, но за останалите референтни типове (например масиви или класове) той работи като ги сравнява по адрес.

Повече за класа **`String`** и за сравняването на символните низове ще научите в главата "Символни низове".

Логически оператори

Да си припомним логическите оператори в C#, тъй като те често се ползват при съставянето на логически (булеви) изрази. Това са операторите: **`&&`**, **`||`**, **`!`** и **`^`**.

Логически оператори && и ||

Логическите оператори **`&&`** (логическо И) и **`||`** (логическо ИЛИ) се използват само върху булеви изрази (стойности от тип **`bool`**). За да бъде резултатът от сравняването на два израза с оператор **`&&`** **`true`** (истина), то и двата операнда трябва да имат стойност **`true`**. Например:

```
bool result = (2 < 3) && (3 < 4);
```

Този израз е "истина", защото и двата операнда: (2 < 3) и (3 < 4) са "истина". Логическият оператор **`&&`** се нарича още и съкратен оператор, тъй като той не губи време за допълнителни изчисления. Той изчислява лявата част на израза (първи операнд) и ако резултатът е **`false`**, не губи време за изчисляването на втория операнд, тъй като е невъзможно крайният резултат да е "истина", ако първият операнд не е "истина". По тази причина той се нарича още **съкратен логически оператор "и"**.

Аналогично операторът || връща дали поне единият операнд от двата има стойност "истина". Пример:

```
bool result = (2 < 3) || (1 == 2);
```

Този израз е "истина", защото първият му операнд е "истина". Както и при `&&` оператора, изчислението се извършва съкратено – ако първият операнд е `true`, вторият изобщо не се изчислява, тъй като резултатът е вече известен. Той се нарича още **съкратен логически оператор "или"**.

Логически оператори & и |

Операторите за сравнение `&` и `|` са подобни, съответно на `&&` и `||`. Разликата се състои във факта, че се изчисляват и двата операнда един след друг, независимо от това, че крайния резултат е предварително ясен. Затова и тези оператори за сравнение се наричат още **несъкратени логически оператори** и се ползват много рядко.

Например, когато се сравняват два операнда с `&` и първият операнд се сведе до "лъжа", въпреки това се продължава с изчисляването на втория операнд. Резултатът е ясно, че ще бъде сведен до "лъжа". По същия начин, когато се сравняват два операнда с `|` и първия операнд се сведе до "истина", независимо от това се продължава с изчисляването на втория операнд и резултатът въпреки всичко се свежда до "истина".

Не трябва да бъркате булевите оператори `&` и `|` с побитовите оператори `&` и `|`. Макар и да се изписват по еднакъв начин, те приемат различни аргументи (съответно булеви изрази или целочислени изрази) и връщат различен резултат (`bool` или цяло число) и действията им не са съвсем идентични.

Логически оператори ^ и !

Операторът `^`, известен още като **изключващо ИЛИ (XOR)**, се прилага само върху булеви стойности. Той се причислява към несъкратените оператори, поради факта, че изчислява и двата операнда един след друг. Резултатът от прилагането на оператора е "истина", когато само и точно един от операндите е истина, но не и двата едновременно. В противен случай резултатът е "лъжа". Ето един пример:

```
Console.WriteLine("Изключващо ИЛИ: " + ((2 < 3) ^ (4 > 3)));
```

Резултатът е следният:

```
Изключващо ИЛИ: False
```

Предходният израз е сведен до лъжа, защото и двата операнда: (2 < 3) и (4 > 3) са истина.

Операторът `!` връща като резултат противоположната стойност на булевия израз, към който е приложен. Пример:

```
bool value = !(7 == 5); // True
Console.WriteLine(value);
```

Горният израз може да бъде прочетен, като "обратното на истинността на израза "**`7 == 5`**". Резултатът от примера е **`True`** (обратното на **`False`**).

Условни конструкции if и if-else

След като си припомнихме как можем да сравняваме изрази, нека преминем към условните конструкции, които ни позволяват да имплементираме програмна логика.

Условните конструкции `if` и `if-else` представляват тип условен контрол, чрез който програмата може да се държи различно, в зависимост от някакво условие, което се проверява по време на изпълнение на конструкцията.

Условна конструкция if

Основният формат на условната конструкция **`if`** е следният:

```
if (булев израз)
{
  тяло на условната конструкция;
}
```

Форматът включва: **`if`**-клауза, булев израз и тяло на условната конструкция.

Булевият израз може да бъде променлива от булев тип или булев логически израз. Булевият израз не може да бъде цяло число (за разлика от други езици за програмиране като C и C++).

Тялото на конструкцията е онази част, заключена между големите къдрави скоби: **`{}`**. То може да се състои от един или няколко операции (statements). Когато се състои от няколко операции, говорим за съставен блоков оператор, т.е. поредица от команди, следващи една след друга, заградени във фигурни скоби.

Изразът в скобите след ключовата дума **`if`** трябва да бива изчислен до булева стойност **`true`** или **`false`**. Ако изразът бъде изчислен до стойност **`true`**, тогава се изпълнява тялото на условната конструкция. Ако резултатът от изчислението на булевия израз е **`false`**, то операторите в тялото няма да бъдат изпълнени.

Условна конструкция if – пример

Да разгледаме един пример за използване на условна конструкция **`if`**:

```
static void Main()
{
   Console.WriteLine("Enter two numbers.");
   Console.Write("Enter first number: ");
   int firstNumber = int.Parse(Console.ReadLine());
   Console.Write("Enter second number: ");
   int secondNumber = int.Parse(Console.ReadLine());
   int biggerNumber = firstNumber;
   if (secondNumber > firstNumber)
   {
      biggerNumber = secondNumber;
   }
   Console.WriteLine("The bigger number is: {0}", biggerNumber);
}
```

Ако стартираме примера и въведем числата 4 и 5, ще получим следния резултат:

```
Enter two numbers.
Enter first number: 4
Enter second number: 5
The bigger number is: 5
```

Конструкцията if и къдравите скоби

При наличието на само един оператор в тялото на **`if`**-конструкцията, къдравите скоби, обозначаващи тялото на условния оператор могат да бъдат изпуснати, както е показано по-долу. Добра практика е, обаче те да бъдат поставяни, дори при наличието на само един оператор. Целта е програмният код да бъде по-четим.

Ето един пример, в който изпускането на къдравите скоби води до объркване:

```
int a = 6;
if (a > 5)
   Console.WriteLine("Променливата а е по-голяма от 5.");
   Console.WriteLine("Този код винаги ще се изпълни!");
// Bad practice: unreadable code
```

В горния пример кодът е форматиран заблуждаващо и създава впечатление, че и двете печатания по конзолата се отнасят за тялото на **`if`** блока, а всъщност това е вярно само за първия от тях.

Винаги слагайте къдрави скоби { } за тялото на if блоковете, дори ако то се състои само от един оператор!

Условна конструкция if-else

В C#, както и в повечето езици за програмиране, съществува условна конструкция с **`else`** клауза: конструкцията **`if-else`**. Нейният формат е, както следва:

```
if (булев израз)
{
  тяло на условната конструкция;
}
else
{
  тяло на else-конструкция;
}
```

Форматът на **`if-else`** конструкцията включва: запазена дума **`if`**, булев израз, тяло на условната конструкция, запазена дума **`else`**, тяло на **`else`**-конструкция. Тялото на **`else`**-конструкцията може да се състои от един или няколко оператора, заградени в къдрави скоби, също както тялото на условната конструкция.

Тази конструкция работи по следния начин: изчислява се изразът в скобите (булевият израз). Резултатът от изчислението трябва да е булев – **`true`** или **`false`**. В зависимост от резултата са възможни два пътя, по които да продължи потока от изчисленията. Ако булевият израз се изчисли до **`true`**, се изпълнява тялото на условната конструкция, а тялото на **`else`**-конструкцията се пропуска и операторите в него не се изпълняват. В обратния случай, ако булевият израз се изчисли до **`false`**, се изпълнява тялото на **`else`**-конструкцията, а основното тяло на условната конструкция се пропуска и операторите в него не се изпълняват.

Условна конструкция if-else – пример

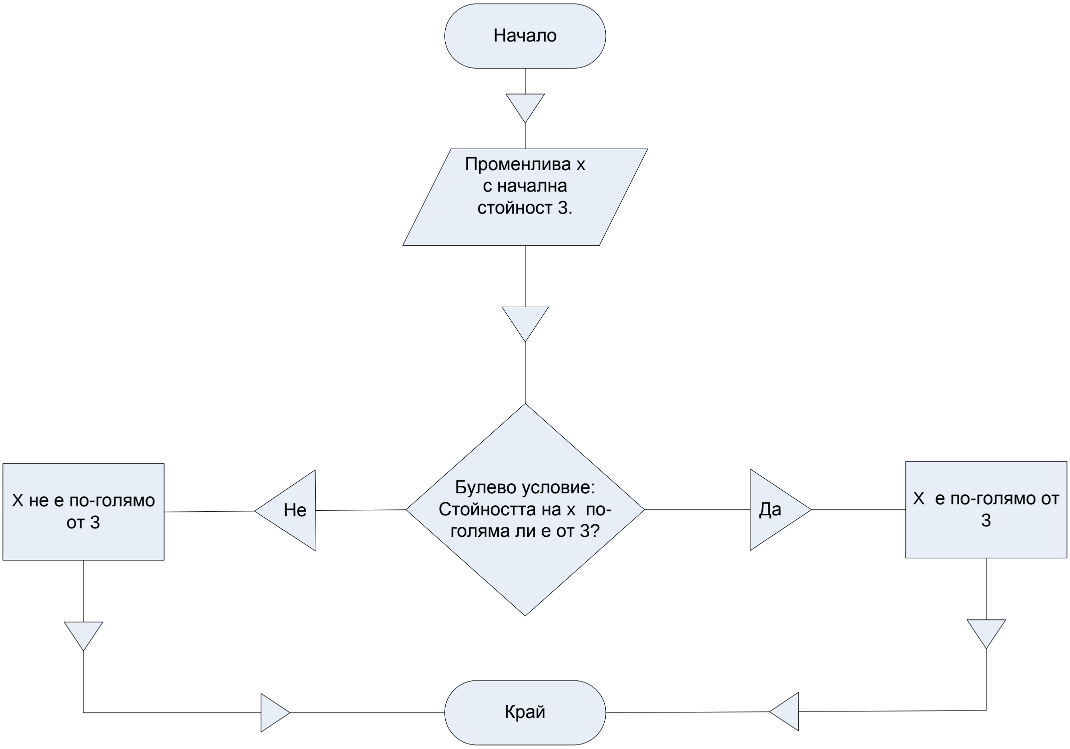
Нека разгледаме следния пример, за да покажем в действие как работи **`if-else`** конструкцията:

```
static void Main()
{
  int x = 2;
  if (x > 3)
  {
    Console.WriteLine("x е по-голямо от 3");
  }
  else
  {
    Console.WriteLine("x не е по-голямо от 3");
  }
}
```

Програмният код може да бъде интерпретиран по следния начин: ако x>3, то резултатът на изхода е: "**x е по-голямо от 3**", иначе (`else`) резултатът е: "**x не е по-голямо от 3**". В случая, понеже x=2, след изчислението на булевия израз ще бъде изпълнен операторът от `else`-конструкцията. Резултатът от примера е:

```
x не е по-голямо от 3
```

На следващата блок-схема е показан графично потокът на изчисленията от този пример:


Начало
Променлива x с начална стойност 3.
Булево условие: Стойността на x по-голяма ли е от 3?
Не
Да
X не е по-голямо от 3
X е по-голямо от 3
Край

Вложени if конструкции

Понякога е нужно програмната логика в дадена програма или приложение да бъде представена посредством `if`-конструкции, които се съдържат една в друга. Наричаме ги **вложени** `if` или `if-else` конструкции.

Влагане наричаме поставянето на `if` или `if-else` конструкция в тялото на друга `if` или `else` конструкция. В такива ситуации всяка `else` клауза се отнася за най-близко разположената предходна `if` клауза. По този начин разбираме коя `else` клауза към коя `if` клауза се отнася.

Не е добра практика нивото на влагане да бъде повече от три, тоест не трябва да бъдат влагани повече от три условни конструкции една в друга. Ако поради една или друга причина се наложи да бъде направено влагане на повече от три конструкции, то част от кода трябва да се изнесе в отделен метод (вж. главата Методи).

Вложени if конструкции – пример

Следва пример за употреба на вложени `if` конструкции:

```
int first = 5;
int second = 3;

if (first == second)
{
  Console.WriteLine("These two numbers are equal.");
}
else
{
  if (first > second)
  {
    Console.WriteLine("The first number is greater.");
  }
  else
  {
    Console.WriteLine("The second number is greater.");
  }
}
```

В примера се разглеждат две числа и се сравняват на две стъпки: първо се сравняват дали са равни и ако не са, се сравняват отново, за да се установи кое от числата е по-голямо. Ето го и резултата от работата на горния код:

```
The first number is greater.
```

Поредици if-else-if-else-...

Понякога се налага да ползваме поредица от `if` конструкции, в `else` клаузата на които има нова `if` конструкция. Ако ползваме вложени `if` конструкции, кодът ще се отмести прекалено навътре. Затова в такива ситуации е допустимо след `else` веднага да следва нов `if`, дори е добра практика. Ето един пример:

```
char ch = 'X';
if (ch == 'A' || ch == 'a')
{
  Console.WriteLine("Vowel [ei]");
}
else if (ch == 'E' || ch == 'e')
{
  Console.WriteLine("Vowel [i:]");
}
else if (ch == 'I' || ch == 'i')
{
  Console.WriteLine("Vowel [ai]");
}
else if (ch == 'O' || ch == 'o')
{
  Console.WriteLine("Vowel [ou]");
}
else if (ch == 'U' || ch == 'u')
{
  Console.WriteLine("Vowel [ju:]");
}
else
{
  Console.WriteLine("Consonant");
}
```

Програмната логика от примера последователно сравнява дадена променлива, за да провери дали тя е някоя от гласните букви на латинската азбука. Всяко следващо сравнение се извършва само в случай че предходното сравнение не е било истина. В крайна сметка, ако никое от **`if`** условията не е изпълнено, се изпълнява последната **`else`** клауза, заради което резултатът от примера е следният:

```
Consonant
```

If конструкции – добри практики

Ето и някои съвети, които е препоръчително да бъдат следвани при писането на **`if`** конструкции:

- Използвайте блокове, заградени с къдрави скоби **`{ }`** след **`if`** и **`else`** с цел избягване на двусмислие.
- Винаги форматирайте коректно програмния код чрез отместване на кода след **`if`** и след **`else`** с една табулация навътре, с цел да бъде лесно четим и да не позволява двусмислие.
- Предпочитайте използването на **`switch-case`** конструкция вместо поредица **`if-else-if-else-…`** конструкции или серия вложени **`if-else`** конструкции, когато това е възможно. Конструкцията **`switch-case`** ще разгледаме в следващата секция.

Условна конструкция switch-case

В следващата секция ще бъде разгледана условната конструкция **`switch`** за избор измежду списък от възможности.

Как работи switch-case конструкцията?

Конструкцията **switch-case** избира измежду части от програмен код на базата на изчислената стойност на определен израз (най-често целочислен). Форматът на конструкцията за избор на вариант е следният:

```
switch (селектор)
{
  case целочислена-стойност-1: конструкция; break;
  case целочислена-стойност-2: конструкция; break;
  case целочислена-стойност-3: конструкция; break;
  case целочислена-стойност-4: конструкция; break;
  // …
  default: конструкция; break;
}
```

Селекторът е израз, връщащ като резултат някаква стойност, която може да бъде сравнявана, например число или **string**. Операторът **switch** сравнява резултата от селектора с всяка една стойност от изброените в тялото на switch конструкцията в **case** етикетите. Ако се открие съвпадение с някой **case** етикет, се изпълнява съответната конструкция (проста или съставна). Ако не се открие съвпадение, се изпълнява **default** конструкцията (когато има такава). Стойността на селектора трябва задължително да бъде изчислена преди да се сравнява със стойностите вътре в **switch** конструкцията. Етикетите не трябва да имат една и съща стойност.

Както се вижда, че в горната дефиниция всеки **case** завършва с оператора **break**, което води до преход към края на тялото на **switch** конструкцията. C# компилаторът задължително изисква да се пише **break** в края на всяка **case**-секция, която съдържа някакъв код. Ако след дадена **case**-конструкция липсва програмен код, **break** може да бъде пропуснат и тогава изпълнението преминава към следващата **case**-конструкция и т.н. до срещането на оператор **break**. След **default** конструкцията, **break** е задължителен.

Не е задължително **default** конструкцията да е на последно място, но е препоръчително да се постави накрая, а не в средата на **switch** конструкцията.

Правила за израза в switch

Конструкцията **switch** е един ясен начин за имплементиране на избор между множество варианти (тоест, избор между няколко различни пътища за изпълнение на програмния код). Тя изисква селектор, който се изчислява до някаква конкретна стойност. Типът на селектора може да бъде цяло число, **string** или **enum**. Ако искаме да използваме, например, низ или число с плаваща запетая като селектор, това няма да работи в

switch конструкция. За нецелочислени типове данни трябва да използваме последователност от **if** конструкции.

Използване на множество етикети

Използването на множество етикети е удачно, когато искаме да бъде изпълнена една и съща конструкция в повече от един от случаите. Нека разгледаме следния пример:

```
int number = 6;
switch (number)
{
   case 1:
   case 4:
   case 6:
   case 8:
   case 10:
      Console.WriteLine("Числото не е просто!"); break;
   case 2:
   case 3:
   case 5:
   case 7:
      Console.WriteLine("Числото е просто!"); break;
   default:
      Console.WriteLine("Не знам какво е това число!"); break;
}
```

В този пример е имплементирано използването на множество етикети чрез **case** конструкции без **break** след тях, така че в случая първо ще се изчисли целочислената стойност на селектора – тук тя е **6**, и след това тази стойност ще започне да се сравнява с всяка една целочислена стойност в **case** конструкциите. След срещане на съвпадение ще бъде изпълнен блокът с кода след съвпадението. Ако съвпадение не бъде срещнато, ще бъде изпълнен **default** блокът. Резултатът от горния пример следният:

```
Числото не е просто!
```

Добри практики при използване на switch-case

- Добра практика при използването на конструкцията за избор на вариант **switch** е **default** конструкцията да бъде поставяна на последно място, с цел програмния код да бъде по-лесно четим.
- Добре е на първо място да бъдат поставяни онези **case** случаи, които обработват най-често възникващите ситуации. **case** конструкциите, които обработват ситуации, възникващи по-рядко могат да бъдат поставени в края на конструкцията.

- Ако стойностите в **`case`** етикетите са целочислени, е препоръчително да се подреждат по големина в нарастващ ред.
- Ако стойностите в **`case`** етикетите са от символен тип, е препоръчително **`case`** етикетите да бъдат подреждани по азбучен ред.
- Препоръчва се винаги да се използва **`default`** блок за прихващане на ситуации, които не могат да бъдат обработени при нормално изпълнение на програмата. Ако при нормалната работа на програмата не се достига до **`default`** блока, в него може да се постави код, който съобщава за грешка.

Упражнения

1. Да се напише **`if`**-конструкция, която проверява стойността на две целочислени променливи и разменя техните стойности, ако стойността на първата променлива е по-голяма от втората.
2. Напишете програма, която показва знака (**+** или **-**) от произведението на три реални числа, без да го пресмята. Използвайте последователност от **`if`** оператори.
3. Напишете програма, която намира най-голямото по стойност число, измежду три дадени числа.
4. Сортирайте 3 реални числа в намаляващ ред. Използвайте вложени **`if`** оператори.
5. Напишете програма, която за дадена цифра (0-9), зададена като вход, извежда името на цифрата на български език.
6. Напишете програма, която при въвеждане на коефициентите (**a**, **b** и **c**) на квадратно уравнение: **ax^2+bx+c**, изчислява и извежда неговите реални корени (ако има такива). Квадратните уравнения могат да имат 0, 1 или 2 реални корена.
7. Напишете програма, която намира най-голямото по стойност число измежду дадени 5 числа.
8. Напишете програма, която по избор на потребителя прочита от конзолата променлива от тип **`int`**, **`double`** или **`string`**. Ако променливата е **`int`** или **`double`**, трябва да се увеличи с 1. Ако променливата е **`string`**, трябва да се прибави накрая символа "*". Отпечатайте получения резултат на конзолата. Използвайте **`switch`** конструкция.
9. Дадени са пет цели числа. Напишете програма, която намира онези подмножества от тях, които имат сума 0. Примери:
    - Ако са дадени числата {3, -2, 1, 1, 8}, сумата на -2, 1 и 1 е 0.
    - Ако са дадени числата {3, 1, -7, 35, 22}, няма подмножества със сума 0.

10. Напишете програма, която прилага бонус точки към дадени точки в интервала [1..9] чрез прилагане на следните правила:
    - Ако точките са между 1 и 3, програмата ги умножава по 10.
    - Ако точките са между 4 и 6, ги умножава по 100.
    - Ако точките са между 7 и 9, ги умножава по 1000.
    - Ако точките са 0 или повече от 9, се отпечатва съобщение за грешка.

11. * Напишете програма, която преобразува дадено число в интервала [0..999] в текст, съответстващ на българското произношение на числото. Примери:
    - 0 → "Нула"
    - 12 → "Дванадесет"
    - 98 → "Деветдесет и осем"
    - 273 → "Двеста седемдесет и три"
    - 400 → "Четиристотин"
    - 501 → "Петстотин и едно"
    - 711 → "Седемстотин и единадесет"

Решения и упътвания

1. Погледнете секцията за if конструкции.

2. Множество от ненулеви числа имат положително произведение, ако отрицателните сред тях са четен брой. Ако отрицателните числа в множеството са нечетен брой, произведението е отрицателно. Ако някое от числата е нула, произведението е нула.

3. Можете да използвате вложени `if` конструкции.

4. Първо намерете най-малкото от трите числа, след това го разменете с първото. После проверете дали второто е по-голямо от третото и ако е така, ги разменете.

5. Най-подходящо е да използвате `switch` конструкция.

6. От математиката е известно, че едно квадратно уравнение може да има един или два реални корена или въобще да няма реални корени. За изчисляване на реалните корени на дадено квадратно уравнение първо се намира стойността на дискриминантата (D) по следната формула: $D=\sqrt{b^2-4ac}$. Ако стойността на дискриминантата е нула, то квадратното уравнение има един двоен реален корен и той се изчислява по следната формула: $x_{1,2}=\frac{-b}{2a}$. Ако стойността на дискриминантата е положително число, то уравнението има два различни реални корени, които се изчисляват по формулата:

$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. Ако стойността на дискриминантата е отрицателно число, то квадратното уравнение няма реални корени.

7. Използвайте вложени `if` конструкции. Можете да използвате конструкцията за цикъл `for`, за която можете да прочетете в следващите глави на книгата или в Интернет.
8. Използвайте входна променлива, която да показва от какъв тип ще е входа, т.е. при въвеждане на 0 типа е `int`, при 1 е `double` и при 2 е `string`.
9. Използвайте вложени `if` конструкции или последователност от сравнения, за да проверите сумите на всичките 15 подмножества на дадените числа (без празното).
10. Използвайте `switch` конструкция и накрая изведете като резултат на конзолата пресметнатите точки.
11. Използвайте вложени `switch` конструкции. Да се обърне специално внимание на числата от 0 до 19 и на онези, в които единиците са 0.

Telerik Academy

Присъедини се към Академията на Телерик!

АКАДЕМИЯТА НА ТЕЛЕРИК предоставя безплатно практическо обучение, насочено към всички млади хора, желаещи да станат умели .NET софтуерни инженери и да се присъединят към екипа на Телерик.

Академията подготвя бъдещите софтуерни специалисти за сложността, разнообразието и динамиката на днешната ИТ действителност, като за целта всички курсисти се обучават да работят с най-новите технологии на Майкрософт и черпят знания и опит от доказали се в областта на програмирането лектори и разработчици на софтуер.

В академията ще получите задълбочени знания и опит, изучавайки:

C#, .NET технологии, бази данни, SQL, Silverlight, WPF, ASP.NET, HTML5, разработка на мобилни приложения за iOS, Android и Windows Phone, основите на софтуерното инженерство

Академията на Телерик ви дава възможност да:

- **Учите напълно БЕЗПЛАТНО**
- **Изберете сред редица РАЗЛИЧНИ КУРСОВЕ**
- **Овладеете ОСНОВИТЕ на софтуерното инженерство**
- **Усвоите ПРОЦЕСА за разработка на софтуер**
- **Получите задълбочени теоретични и практически ИТ ПОЗНАНИЯ**
- **Станете умел .NET СОФТУЕРЕН ИНЖЕНЕР**
- **Започнете своята ИТ кариера в ТЕЛЕРИК – РАБОТОДАТЕЛ #1 в България за 2010 г.**

Само в рамките на две години **АКАДЕМИЯТА НА ТЕЛЕРИК** за софтуерни инженери успя да се наложи като безспорен лидер у нас в предлагането на допълнително обучение за софтуерни специалисти, спомагайки за успешния старт в кариерното развитие на стотици ентусиазирани младежи.

academy.telerik.com

academy@telerik.com *facebook.com/TelerikAcademy*

telerik
deliver more than expected

Глава 6. Цикли

В тази тема...

В настоящата тема ще разгледаме конструкциите за цикли, с които можем да изпълняваме даден фрагмент програмен код многократно. Ще разгледаме как се реализират повторения с условие (**`while`** и **`do-while`** цикли) и как се работи с **`for`**-цикли. Ще дадем примери за различните възможности за дефиниране на цикъл, за начина им на конструиране и за някои от основните им приложения. Накрая ще разгледаме, как можем да използваме няколко цикъла, разположени един в друг (вложени цикли).

Какво е "цикъл"?

В програмирането често се налага многократно изпълнение на дадена последователност от операции. **Цикъл** (loop) е основна конструкция в програмирането, която позволява многократно изпълнение на даден фрагмент сорс код. В зависимост от вида на цикъла, програмният код в него се повтаря или фиксиран брой пъти или докато е в сила дадено условие.

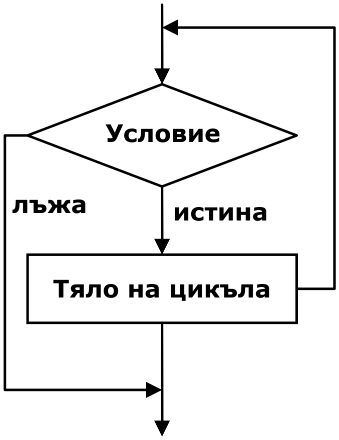
Цикъл, който никога не завършва, се нарича **безкраен цикъл** (infinite loop). Използването на безкраен цикъл рядко се налага, освен в случаи, когато някъде в тялото на цикъла се използва операторът **`break`**, за да бъде прекратено неговото изпълнение преждевременно. Ще разгледаме тази възможност по-късно, а сега нека разгледаме конструкциите за цикли в езика C#.

Конструкция за цикъл while

Един от най-простите и най-често използвани цикли е **`while`**.

```
while (условие)
{
  тяло на цикъла;
}
```

В кода по-горе **`условие`** представлява произволен израз, който връща булев резултат – истина (**`true`**) или лъжа (**`fasle`**). Той определя докога ще се изпълнява тялото на цикъла и се нарича **условие на цикъла** (loop condition). В примера **`тяло на цикъла`** е програмният код, изпълняван при всяко повторение (итерация) на цикъла, т.е. всеки път, когато входното условие е истина. Логически поведението на **`while`** циклите може да се опише чрез следната схема:



При **`while`** цикъла първоначално се изчислява булевият израз и ако резултатът от него е **`true`**, се изпълнява последователността от операции в тялото на цикъла. След това входното условие отново се проверява и ако е истина, отново се изпълнява тялото на цикъла. Всичко това се повтаря отново и отново докато в някакъв момент условният израз върне стойност **`false`**. В този момент цикълът приключва своята работа и програмата продължава от следващия ред веднага след тялото на цикъла.

Понеже условието на **`while`** цикъла се намира в неговото начало, той често се нарича цикъл с предусловие (**pre-test loop**). Тялото на **`while`** цикъл може и да не се изпълни нито веднъж, ако в самото начало е нарушено условието на цикъла. Ако условието на цикъла никога не бъде нарушено, той ще се изпълнява безкрайно.

Използване на while цикли

Нека разгледаме един съвсем прост пример за използването на **`while`** цикъл. Целта на цикъла е да се отпечатват на конзолата числата в интервала от 0 до 9 в нарастващ ред:

```
// Initialize the counter
int counter = 0;

// Execute the loop body while the loop condition holds
while (counter <= 9)
{
   // Print the counter value
   Console.WriteLine("Number : " + counter);
   // Increment the counter
   counter++;
}
```

При изпълнение на примерния код получаваме следния резултат:

```
Number : 0
Number : 1
Number : 2
Number : 3
Number : 4
Number : 5
Number : 6
Number : 7
Number : 8
Number : 9
```

Нека дадем още примери, за да илюстрираме ползата от циклите и покажем някои задачи, които могат да се решават с помощта на цикли.

Сумиране на числата от 1 до N – пример

В настоящия пример ще разгледаме как с помощта на цикъл **`while`** можем да намерим сумата на числата от **`1`** до **`n`**. Числото **`n`** се чете от конзолата:

```
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());
int num = 1;
int sum = 1;
Console.Write("The sum 1");
while (num < n)
{
   num++;
   sum += num;
   Console.Write(" + " + num);
}
Console.WriteLine(" = " + sum);
```

Първоначално инициализираме променливите **num** и **sum** със стойност **1**. В **num** пазим текущото число, което добавяме към сумата на предходните. При всяко преминаване през цикъла увеличаваме **num** с **1**, за да получим следващото число, след което в условието на цикъла проверяваме дали то е в интервала от **1** до **n**. Променливата **sum** съдържа сумата на числата от 1 до **num** във всеки един момент. При влизане в цикъла добавяме към нея поредното число записано в **num**. На конзолата принтираме всички числа **num** от 1 до n с разделител "**+**" и крайния резултат от сумирането след приключване на цикъла. Изходът от програмата е следният (въвеждаме n=17):

```
n = 17
The sum 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 +
14 + 15 + 16 + 17 = 153
```

Нека дадем още един пример за използване на **while**, преди да продължим към другите конструкции за организиране на цикъл.

Проверка за просто число – пример

Ще напишем програма, с която да проверяваме дали дадено число е просто. Числото за проверка ще четем от конзолата. Както знаем от математиката, просто е всяко цяло положително число, което освен на себе си и на 1, не се дели на други числа. Можем да проверим дали числото **num** е просто, като в цикъл проверим дали се дели на всички числа между 2 и √**num**:

```
Console.Write("Enter a positive number: ");
int num = int.Parse(Console.ReadLine());
int divider = 2;
int maxDivider = (int)Math.Sqrt(num);
bool prime = true;
while (prime && (divider <= maxDivider))
{
   if (num % divider == 0)
```

```
    {
        prime = false;
    }
    divider++;
}
Console.WriteLine("Prime? " + prime);
```

Променливата **`divider`** използваме за стойността на евентуалния делител на числото. Първоначално я инициализираме с 2 (най-малкият възможен делител). Променливата **`maxDivider`** е максималният възможен делител, който е равен на корен квадратен от числото. Ако имаме делител, по-голям от √**`num`**, то би трябвало **`num`** да има и друг делител, който е обаче по-малък от √**`num`** и затова няма смисъл да проверяваме числата, по-големи от √**`num`**. Така намаляваме броя на итерациите на цикъла.

За резултата използваме отделна променлива от булев тип с име **`prime`**. Първоначално, нейната стойност е **`true`**. При преминаване през цикъла, ако се окаже, че числото има делител, стойността на **`prime`** ще стане **`false`**.

Условието на **`while`** цикъла се състои от две подусловия, които са свързани с логическия оператор **`&&`** (логическо и). За да бъде изпълнен цикълът, трябва и двете подусловия да са верни едновременно. Ако в някакъв момент намерим делител на числото **`num`**, променливата **`prime`** става **`false`** и условието на цикъла вече не е изпълнено. Това означава, че цикълът се изпълнява до намиране на първия делител на числото или до доказване на факта, че **`num`** не се дели на никое от числата в интервала от 2 до √**`num`**.

Ето как изглежда резултатът от изпълнението на горния пример при въвеждане съответно на числата 37 и 34 като входни стойности:

```
Enter a positive number: 37
Prime? True
```

```
Enter a positive number: 34
Prime? False
```

Оператор break

Операторът **`break`** се използва за преждевременно излизане от цикъл, преди той да е завършил изпълнението си по естествения си начин. При срещане на оператора **`break`** цикълът се прекратява и изпълнението на програмата продължава от следващия ред веднага след тялото на цикъла. Прекратяването на цикъл с оператора **`break`** може да стане само от неговото тяло, когато то се изпълнява в поредната итерация на цикъла. Когато **`break`** се изпълни кодът след него в тялото на цикъла се прескача и не се изпълнява. Ще демонстрираме аварийното излизане от цикъл с **`break`** с един пример.

Изчисляване на факториел – пример

В този пример ще пресметнем факториела на въведено от конзолата число с помощта на безкраен **while** цикъл и оператора **break**. Да си припомним от математиката какво е факториел и как се изчислява. Факториелът на дадено естествено число n е функция, която изчислява като произведение на всички естествени числа, по-малки или равни на n. Записва се като n! и по дефиниция са в сила формулите:

- n! = 1*2*3…….(n-1)*n, за n>1;
- 1! = 1;
- 0! = 1.

Произведението n! може да се изрази чрез факториел от естествени числа, по-малки от n:

- n! = (n-1)! * n, като използваме началната стойност 0! = 1.

За да изчислим факториела на n ще използваме директно дефиницията:

```
int n = int.Parse(Console.ReadLine());
// "decimal" is the biggest type that can hold integer values
decimal factorial = 1;
// Perform an "infinite loop"
while (true)
{
   if (n <= 1)
   {
      break;
   }
   factorial *= n;
   n--;
}
Console.WriteLine("n! = " + factorial);
```

В началото инициализираме променливата **factorial** с 1, а n прочитаме от конзолата. Конструираме безкраен **while** цикъл като използваме **true** за условие на цикъла. Използваме оператора **break**, за да прекратим цикъла, когато n достигне стойност по-малка или равна на 1. В противен случай умножаваме текущия резултат по n и намаляваме n с единица. Така на практика първата итерация от цикъла променливата **factorial** има стойност n, на втората – n*(n-1) и т.н. На последната итерация от цикъла стойността на **factorial** е произведението n*(n-1)*(n-2)*…*3*2, което е търсената стойност n!.

Ако изпълним примерната програма и въведем 10 като вход, ще получим следния резултат:

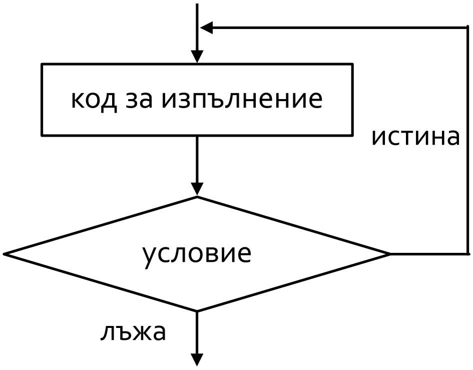
```
10
n! = 3628800
```

Конструкция за цикъл do-while

Do-while цикълът е аналогичен на `while` цикъла, само че при него проверката на булевото условие се извършва след изпълнението на операциите в цикъла. Този тип цикли се наричат цикли с условие в края (post-test loop). Един `do-while` цикъл изглежда по следния начин:

```
do
{
   код за изпълнение;
}
while (израз);
```

Схематично `do-while` циклите се изпълняват по следната логическа схема:



Първоначално се изпълнява тялото на цикъла. След това се проверява неговото условие. Ако то е истина, тялото на цикъла се повтаря, а в противен случай цикълът завършва. Тази логика се повтаря докато условието на цикъла бъде нарушено. Тялото на цикъла се повтаря най-малко един път. Ако условието на цикъла постоянно е истина, цикълът никога няма да завърши.

Използване на do-while цикли

Do-while цикълът се използва, когато искаме да си гарантираме, че поредицата от операции в него ще бъде изпълнена многократно и задължително поне веднъж в началото на цикъла.

Изчисляване на факториел – пример

В този пример отново ще изчислим факториела на дадено число n, но този път вместо безкраен `while` цикъл, ще използваме `do-while`. Логиката е аналогична на тази в предходния пример:

```
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());
```

```
decimal factorial = 1;
do
{
  factorial *= n;
  n--;
} while (n > 0);
Console.WriteLine("n! = " + factorial);
```

Започваме в началото от резултат 1 и умножаваме последователно на всяка итерация резултата с n и намаляваме n с единица докато n достигне 0. Така получаваме произведението n*(n-1)*…*1. Накрая отпечатваме получения резултат на конзолата. Този алгоритъм винаги извършва поне 1 умножение и затова няма да работи коректно при n=0, но ще работи правилно за n ≥ 1.

Ето го и резултатът от изпълнение на горния пример при n=7:

```
n = 7
n! = 5040
```

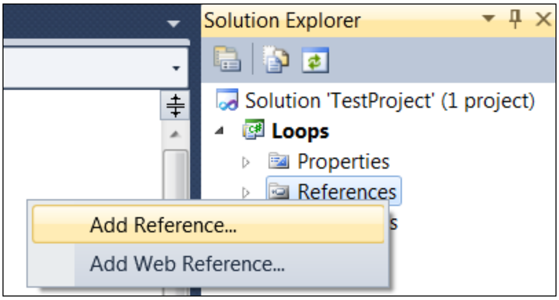
Факториел на голямо число – пример

Може би се питате какво ще се случи, ако в предходния пример въведем прекалено голяма стойност за числото n, например n=100. Тогава ще при пресмятането на n! ще препълним типа **decimal** и резултатът ще е изключение **System.OverflowException**:

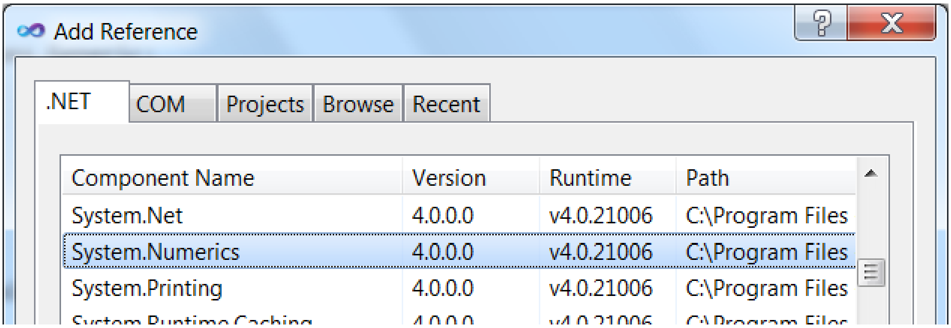
```
n = 100
Unhandled Exception: System.OverflowException: Value was either
too large or too small for a Decimal.
   at System.Decimal.FCallMultiply(Decimal& result, Decimal d1,
Decimal d2)
   at System.Decimal.op_Multiply(Decimal d1, Decimal d2)
   at TestProject.Program.Main() in
C:\Projects\TestProject\Program
.cs:line 17
```

Ако искаме да пресметнем 100!, можем да използваме типа данни **BigInteger**, който е нов за .NET Framework 4.0 и липсва в по-старите версии. Този тип представлява цяло число, което може да бъде много голямо (примерно 100 000 цифри). Няма ограничение за големината на числата записвани в класа **BigInteger** (стига да има достатъчно оперативна памет).

За да използваме **BigInteger**, първо трябва да добавим референция от нашия проект към асемблито **System.Numerics.dll** (това е стандартна .NET библиотека за работа с много големи цели числа). Добавянето на референция става с щракване с десния бутон на мишката върху референциите на текущия проект в прозореца Solution Explorer на Visual Studio:

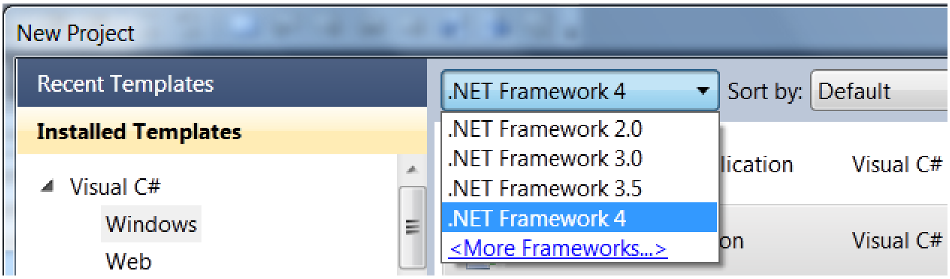

Solution Explorer
Solution 'TestProject' (1 project)
Loops
Properties
References
Add Reference...
Add Web Reference...

Избираме асемблито **`System.Numerics.dll`** от списъка:


Add Reference
.NET
COM
Projects
Browse
Recent

Component Name	Version	Runtime	Path
System.Net	4.0.0.0	v4.0.21006	C:\Program Files
System.Numerics	4.0.0.0	v4.0.21006	C:\Program Files
System.Printing	4.0.0.0	v4.0.21006	C:\Program Files

Ако търсеното асембли липсва в списъка, то Visual Studio проектът вероятно не таргетира .NET Framework 4.0 и трябва или да създадем нов проект или да сменим версията на текущия:


New Project
Recent Templates
Installed Templates
Visual C#
Windows
Web
.NET Framework 4
Sort by:
Default
.NET Framework 2.0
.NET Framework 3.0
.NET Framework 3.5
.NET Framework 4
<More Frameworks...>
lication
Visual C#
on
Visual C#

След това трябва да добавим "**`using System.Numerics;`**" преди началото на класа на нашата програма и да сменим **`decimal`** с **`BigInteger`**. Програмата добива следния вид:

```
using System;
using System.Numerics;

class Factorial
{
   static void Main()
   {
      Console.Write("n = ");
      int n = int.Parse(Console.ReadLine());
      BigInteger factorial = 1;
```

```
     do
     {
       factorial *= n;
       n--;
     } while (n > 0);
     Console.WriteLine("n! = " + factorial);
   }
}
```

Ако сега изпълним програмата за n=100, ще получим стойността на 100 факториел, което е 158-цифрено число:

```
n = 100
n! = 933262154439441526816992388562667004907159682643816214685929
63895217599993229915608941463976156518286253697920827223758251185
210916864000000000000000000000000
```

Произведение в интервала [N…M] – пример

Нека дадем още един, по-интересен, пример за работа с **`do-while`** цикли. Задачата е да намерим произведението на всички числа в интервала [n…m]. Ето едно примерно решение на тази задача:

```
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());
Console.Write("m = ");
int m = int.Parse(Console.ReadLine());
int num = n;
long product = 1;
do
{
  product *= num;
  num++;
}
while (num <= m);
Console.WriteLine("product[n..m] = " + product);
```

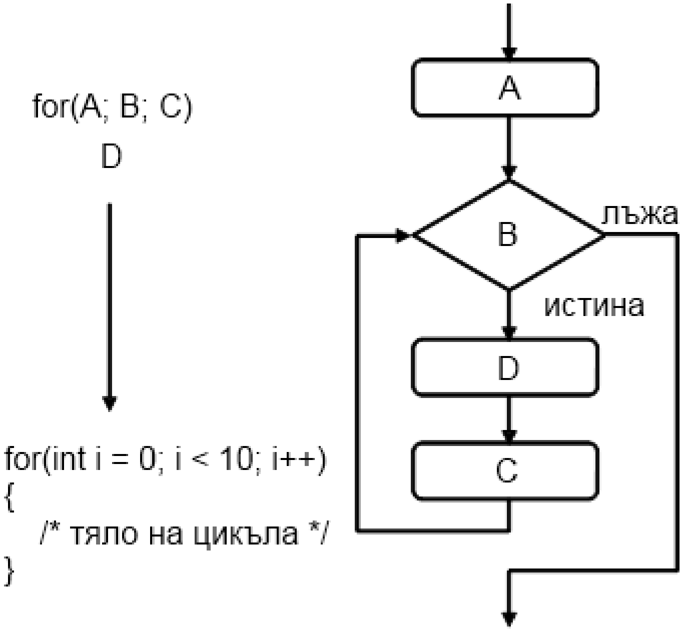
В примерния код на променливата **`num`** присвояваме последователно на всяка итерация на цикъла стойностите n, n+1, …, m и в променливата **`product`** натрупваме произведението на тези стойности. Изискваме от потребителя да въведе n, което да е по-малко от m. В противен случай ще получим като резултат числото n.

Ако стартираме програмата за n=2 и m=6, ще получим следния резултат:

```
n = 2
m = 6
product[n..m] = 720
```

Конструкция за цикъл for

For-циклите са малко по-сложни от **`while`** и **`do-while`** циклите, но за сметка на това могат да решават по-сложни задачи с по-малко код. Ето как изглежда логическата схема, по която се изпълняват **`for`**-циклите:



```
for(int i = 0; i < 10; i++)
{
   /* тяло на цикъла */
}
```

Те съдържат инициализационен блок (A), условие на цикъла (B), тяло на цикъла (D) и команди за обновяване на водещите променливи (C). Ще ги обясним в детайли след малко. Преди това нека разгледаме как изглежда програмният код на един **`for`**-цикъл:

```
for (инициализация; условие; обновяване)
{
  тяло на цикъла;
}
```

Той се състои от инициализационна част за брояча (в схемата **`int i = 0`**), булево условие (**`i < 10`**), израз за обновяване на брояча (**`i++`**, може да бъде **`i--`** или например **`i = i + 3`**) и тяло на цикъла.

Броячът на **`for`** цикъла го отличава от останалите видове цикли. Най-често броячът се променя от дадена начална стойност към дадена крайна стойност в нарастващ ред, примерно от 1 до 100. Броят на итерациите на даден **`for`**-цикъл най-често е известен още преди да започне изпълнението му. Един **`for`**-цикъл може да има една или няколко водещи променливи, които се движат в нарастващ ред или в намаляващ ред или с някаква стъпка. Възможно е едната водеща променлива да расте, а другата – да намалява. Възможно е дори да направим цикъл от 2 до 1024 със стъпка умножение по 2, тъй като обновяването на водещите променливи може да съдържа не само събиране, а всякакви други аритметични операции.

Тъй като никой от изброените елементи на **`for`**-циклите не е задължителен, можем да ги пропуснем всичките и ще получим безкраен цикъл:

```
for ( ; ; )
{
   тяло на цикъла;
}
```

Нека сега разгледаме в детайли отделните части на един **`for`**-цикъл.

Инициализация на for цикъла

For-циклите могат да имат инициализационен блок:

```
for (int num = 0; ...; ...)
{
   // Променливата num е видима тук и може да се използва
}
// Тук num не може да се използва
```

Той се изпълнява само веднъж, точно преди влизане в цикъла. Обикновено инициализационният блок се използва за деклариране на променливата-брояч (нарича се още водеща променлива) и задаване на нейна начална стойност. Тази променлива е "видима" и може да се използва само в рамките на цикъла. Възможно е инициализационният блок да декларира и инициализира повече от една променлива.

Условие на for цикъла

For-циклите могат да имат условие за повторение:

```
for (int num = 0; num < 10; ...)
{
   тяло на цикъла;
}
```

Условието за повторение (**loop condition**) се изпълнява веднъж, преди всяка итерация на цикъла, точно както при **`while`** циклите. При резултат **`true`** се изпълнява тялото на цикъла, а при **`false`** то се пропуска и цикълът завършва (преминава се към останалата част от програмата, веднага след цикъла).

Обновяване на водещата променлива

Последната част от един **`for`**-цикъл съдържа код, който обновява водещата променлива:

```
for (int num = 0; num < 10; num++)
{
  тяло на цикъла;
}
```

Този код се изпълнява след всяка итерация, след като е приключило изпълнението на тялото на цикъла. Най-често се използва за обновяване стойността на брояча.

Тяло на цикъла

Тялото на цикъла съдържа произволен блок със сорс код. В него са достъпни водещите променливи, декларирани в инициализационния блок на цикъла.

For-цикъл – примери

Ето един цялостен пример за `for`-цикъл:

```
for (int i = 0; i <= 10; i++)
{
  Console.Write(i + " ");
}
```

Резултатът от изпълнението му е следният:

```
0 1 2 3 4 5 6 7 8 9 10
```

Ето още един по-сложен пример за `for`-цикъл, в който имаме две водещи променливи **`i`** и **`sum`**, които първоначално имат стойност 1, но ги обновяваме последователно след всяка итерация на цикъла:

```
for (int i = 1, sum = 1; i <= 128; i = i * 2, sum += i)
{
  Console.WriteLine("i={0}, sum={1}", i, sum);
}
```

Резултатът от изпълнението на цикъла е следният:

```
i=1, sum=1
i=2, sum=3
i=4, sum=7
i=8, sum=15
i=16, sum=31
i=32, sum=63
i=64, sum=127
i=128, sum=255
```

Изчисляване на N^M – пример

Като следващ пример ще напишем програма, която повдига числото n на степен m, като за целта ще използваме **`for`**-цикъл:

```
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());
Console.Write("m = ");
int m = int.Parse(Console.ReadLine());
decimal result = 1;
for (int i = 0; i < m; i++)
{
  result *= n;
}
Console.WriteLine("n^m = " + result);
```

Първоначално инициализираме резултата (**`result = 1`**). Цикълът започваме със задаване на начална стойност за променливата-брояч (**`int i = 0`**). Определяме условието за изпълнение на цикъла (**`i < m`**). Така цикълът ще се изпълнява от **`0`** до **`m-1`** или точно **`m`** пъти. При всяко изпълнение на цикъла умножаваме резултата по **`n`** и така **`n`** ще се вдига на поредната степен (1, 2, … m) на всяка итерация. Накрая отпечатаме резултата, за да видим правилно ли работи програмата.

Ето как изглежда изходът от програмата при n=2 и m=10:

```
n = 2
m = 10
n^m = 1024
```

For-цикъл с няколко променливи

Както вече видяхме, с конструкцията за **`for`**-цикъл можем да ползваме едновременно няколко променливи. Ето един пример, в който имаме два брояча. Единият брояч се движи от 1 нагоре, а другият се движи от 10 надолу:

```
for (int small=1, large=10; small<large; small++, large--)
{
  Console.WriteLine(small + " " + large);
}
```

Условието за прекратяване на цикъла е застъпване на броячите. В крайна сметка се получава следният резултат:

```
1 10
2 9
3 8
4 7
```

```
5 6
```

Оператор continue

Операторът **`continue`** спира текущата итерация на най-вътрешния цикъл, без да излиза от него. С помощта на следващия пример ще разгледаме как точно се използва този оператор.

Ще намерим сумата на всички нечетни естествени числа в интервала [1…n], който не се делят на 7 чрез **`for`**-цикъл:

```
int n = int.Parse(Console.ReadLine());
int sum = 0;
for (int i = 1; i <= n; i += 2)
{
  if (i % 7 == 0)
  {
    continue;
  }
  sum += i;
}
Console.WriteLine("sum = " + sum);
```

Първоначално инициализираме водещата променлива на цикъла със стойност 1, тъй като това е първото нечетно естествено число в интервала [1…n]. След всяка итерация на цикъла проверяваме дали **`i`** е все още не е надвишило n (**`i <= n`**). В израза за обновяване на променливата я увеличаваме с 2, за да преминаваме само през нечетните числа. В тялото на цикъла правим проверка дали текущото число се дели на 7. Ако това е изпълнено извикваме оператора **`continue`**, който прескача останалата част от тялото на цикъла (пропуска добавянето текущото число към сумата). Ако числото не се дели на седем, се преминава към обновяване на сумата с текущото число. Резултатът от примера при n=11 е следният:

```
11
sum = 29
```

Конструкция за цикъл foreach

Цикълът **`foreach`** (разширен **`for`**-цикъл) е нов за C/C++/C# фамилията от езици, но е добре познат на VB и PHP програмистите. Тази конструкция служи за обхождане на всички елементи на даден масив, списък или друга колекция от елементи. Подробно с масивите ще се запознаем в темата "Масиви", но за момента можем да си представяме един масив като наредена последователност от числа или други елементи.

Ето как изглежда един **`foreach`** цикъл:

```
foreach (variable in collection)
{
   statements;
}
```

Както виждате, той е значително по-прост от стандартния **`for`**-цикъл и затова много-често се предпочита от програмистите, тъй като спестява писане, когато трябва да се обходят всички елементи на дадена колекция. Ето един пример, който показва как можем да използваме **`foreach`**:

```
int[] numbers = { 2, 3, 5, 7, 11, 13, 17, 19 };
foreach (int i in numbers)
{
   Console.Write(" " + i);
}
Console.WriteLine();
String[] towns = { "Sofia", "Plovdiv", "Varna", "Bourgas" };
foreach (String town in towns)
{
   Console.Write(" " + town);
}
```

В примера се създава масив от числа и след това те се обхождат с **`foreach`** цикъл и се отпечатват на конзолата. След това се създава масив от имена на градове (символни низове) и по същия начин те обхождат и отпечатват на конзолата. Резултатът от примера е следният:

```
2 3 5 7 11 13 17 19
Sofia Plovdiv Varna Bourgas
```

Вложени цикли

Вложените цикли представляват конструкция от няколко цикъла, разположени един в друг. Най-вътрешния цикъл се изпълнява най-много пъти, а най-външният – най-малко. Да разгледаме как изглеждат два вложени цикъла:

```
for (инициализация; проверка; обновяване)
{
   for (инициализация; проверка; обновяване)
   {
      код за изпълнение;
   }
   …
}
```

След инициализация на първия **`for`** цикъл ще започне да се изпълнява неговото тяло, което съдържа втория (вложения) цикъл. Ще се инициализира променливата му, ще се провери условието му и ще се изпълни кода в тялото му, след което ще се обнови променливата му и изпълнението му ще продължи, докато условието му не върне **`false`**. След това ще продължи втората итерация на първия **`for`** цикъл, ще се извърши обновяване на неговата променлива и отново ще бъде изпълнен целия втори цикъл. Вътрешният цикъл ще се изпълни толкова пъти, колкото се изпълнява тялото на външния цикъл.

Нека сега разгледаме един реален пример, с който ще демонстрираме колко полезни са вложените цикли.

Отпечатване на триъгълник – пример

Нека си поставим следната задача: по дадено число n да отпечатаме на конзолата триъгълник с n на брой реда, изглеждащ по следния начин:

```
1
1 2
1 2 3
. . .
1 2 3 . . . n
```

Ще решим задачата с два **`for`**-цикъла. Външният цикъл ще обхожда редовете, а вътрешният – елементите в тях. Когато сме на първия ред, трябва да отпечатаме "1" (1 елемент, 1 итерация на вътрешния цикъл). На втория ред трябва да отпечатаме "1 2" (2 елемента, 2 итерации на вътрешния цикъл). Виждаме, че има зависимост между реда, на който сме и броя на елементите, който ще отпечатваме. Това ни подсказва как да организираме конструкцията на вътрешния цикъл:

- инициализираме водещата му променлива с 1 (първото число, което ще отпечатаме): **`col`** = 1;
- условието за повторение зависи от реда, на който сме: **`col`** <= **`row`**;
- на всяка итерация на вътрешния цикъл увеличаваме с единица водещата променлива.

На практика трябва да направим един **`for`**-цикъл (външен) от 1 до n (за редовете) и в него още един **`for`**-цикъл (вътрешен) за числата в текущия ред, който да върти от 1 до номера на текущия ред. Външният цикъл трябва да ходи по редовете, а вътрешният – по колоните от текущия ред. В крайна сметка получаваме следния код:

```
int n = int.Parse(Console.ReadLine());
for (int row = 1; row <= n; row++)
{
   for (int col = 1; col <= row; col++)
   {
      Console.Write(col + " ");
   }
   Console.WriteLine();
}
```

Ако го изпълним, ще се убедим, че работи коректно. Ето как изглежда резултатът при n=7:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
```

Прости числа в даден интервал – пример

Нека разгледаме още един пример за вложени цикли. Поставяме си за цел да отпечатаме на конзолата всички прости числа в интервала [n, m]. Интервалът ще ограничим с **for**-цикъл, а за проверката за просто число ще използваме вложен **while** цикъл:

```
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());
Console.Write ("m = ");
int m = int.Parse(Console.ReadLine ());
for (int num = n; num <= m; num++)
{
   bool prime = true;
   int divider = 2;
   int maxDivider = (int) Math.Sqrt(num);
   while (divider <= maxDivider)
   {
      if (num % divider == 0)
      {
         prime = false;
         break;
      }
      divider++;
   }
   if (prime)
   {
      Console.Write (" " + num);
   }
}
```

С външния **for**-цикъл проверяваме всяко от числата n, n+1, …, m дали е просто. При всяка итерация на външния **for**-цикъл правим проверка дали водещата му променлива **num** е просто число. Логиката, по която правим проверката за просто число, вече ни е позната. Първоначално инициализираме променливата **prime** с **true**. С вътрешния **while** цикъл проверяваме всяко от числата [2…√**num**] дали е делител на **num** и ако е така, установяваме **prime** във **false**. След завършване на **while** цикъла булевата променлива **prime** показва дали числото е просто или не. Ако поредното число е просто, го отпечатваме на конзолата.

Ако изпълним примера за n=3 и m=75 ще получим следния резултат:

```
n = 3
m = 75
3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73
```

Щастливи числа – пример

Нека разгледаме още един пример, с който ще покажем, че можем да влагаме и повече от два цикъла един в друг. Целта е да намерим и отпечатаме всички четирицифрени числа от вида ABCD, за които: A+B = C+D (наричаме ги щастливи числа). Това ще реализираме с помощта на четири **for**-цикъла – за всяка цифра по един. Най-външният цикъл ще ни определя хилядите. Той ще започва от 1, а останалите от 0. Останалите цикли ще ни определят съответно стотиците, десетиците и единиците. Ще правим проверка дали текущото ни число е щастливо в най-вътрешния цикъл и ако е така, ще го отпечатваме на конзолата. Ето примерна реализация:

```
for (int a = 1; a <= 9; a++)
{
   for (int b = 0; b <= 9; b++)
   {
      for (int c = 0; c <= 9; c++)
      {
         for (int d = 0; d <= 9; d++)
         {
            if ((a + b) == (c + d))
            {
               Console.WriteLine(
                  " " + a + " " + b + " " + c + " " + d);
            }
         }
      }
   }
}
```

Ето част от отпечатания резултат (целият е много дълъг):

```
1 0 0 1
1 0 1 0
1 1 0 2
1 1 1 1
```

```
1 1 2 0
1 2 0 3
1 2 1 2
1 2 2 1
…
```

Оставяме за домашно на старателния читател да предложи по-ефективно решение на същата задача, което ползва само три вложени цикъла, а не четири.

ТОТО 6/49 – пример

В следващия пример ще намерим всички възможни комбинации от тотото в играта "6/49". Трябва да намерим и отпечатаме всички възможни извадки от 6 различни числа, всяко от които е в интервала [1...49]. Ще използваме 6 **`for`**-цикъла. За разлика от предния пример, числата не могат да се повтарят. За да избегнем повторенията ще се стремим всяко следващо число да е по-голямо от предходното. Затова вътрешните цикли няма да започват от 1, а от числото, до което е стигнал предходния цикъл + 1. Първият цикъл ще трябва да го въртим до 44 (а не до 49), вторият до 45, и т.н. Последният цикъл ще е до 49. Ако въртим всички цикли до 49, ще получим съвпадащи числа в някои от комбинациите. По същата причина всеки следващ цикъл започва от брояча на предходния + 1. Нека да видим какво ще се получи:

```
for (int i1 = 1; i1 <= 44; i1++)
{
   for (int i2 = i1 + 1; i2 <= 45; i2++)
   {
      for (int i3 = i2 + 1; i3 <= 46; i3++)
      {
         for (int i4 = i3 + 1; i4 <= 47; i4++)
         {
            for (int i5 = i4 + 1; i5 <= 48; i5++)
            {
               for (int i6 = i5 + 1; i6 <= 49; i6++)
               {
                  Console.WriteLine(i1 + " " + i2 + " " +
                     i3 + " " + i4 + " " + i5 + " " + i6);
               }
            }
         }
      }
   }
}
```

Всичко изглежда правилно. Да стартираме програмата. Изглежда, че работи, но има един проблем – комбинациите са прекален много и програмата не завършва (едва ли някой ще я изчака). Това е в реда на нещата и е една от причините да има ТОТО 6/49 – комбинациите наистина са много. Оставяме за упражнение на любознателния читател да промени горния пример така че само да пребори колко са всичките комбинации от тотото, вместо да ги отпечата. Тази промяна ще намали драстично обемът на отпечатаните на конзолата резултати и програмата ще завърши удивително бързо.

Упражнения

1. Напишете програма, която отпечатва на конзолата числата от 1 до N. Числото N трябва да се чете от стандартния вход.
2. Напишете програма, която отпечатва на конзолата числата от 1 до N, които не се делят едновременно на 3 и 7. Числото N да се чете от стандартния вход.
3. Напишете програма, която чете от конзолата поредица от цели числа и отпечатва най-малкото и най-голямото от тях.
4. Напишете програма, която отпечатва всички възможни карти от стандартно тесте карти без джокери (имаме 52 карти: 4 бои по 13 карти).
5. Напишете програма, която чете от конзолата числото N и отпечатва сумата на първите N члена от редицата на Фибоначи: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...
6. Напишете програма, която пресмята N!/K! за дадени N и K (1<K<N).
7. Напишете програма, която пресмята N!*K!/(N-K)! за дадени N и K (1<K<N).
8. В комбинаториката числата на Каталан (Catalan's numbers) се изчисляват по следната формула: $C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$, за n ≥ 0.

   Напишете програма, която изчислява n-тото число на Каталан за дадено n.
9. Напишете програма, която за дадено цяло число n, пресмята сумата: $S = 1 + {}^{1!}/_{x} + {}^{2!}/_{x^2} + \ldots + {}^{n!}/_{x^n}$
10. Напишете програма, която чете от конзолата положително цяло число N (N < 20) и отпечатва матрица с числа като на фигурата по-долу:

N = 3

1	2	3
2	3	4
3	4	5

N = 4

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

11. Напишете програма, която пресмята с колко нули завършва факториелът на дадено число. Примери:

    N = 10 -> N! = 36288**00** -> 2

    N = 20 -> N! = 243290200817664**0000** -> 4

12. Напишете програма, която преобразува дадено число от десетична в двоична бройна система.
13. Напишете програма, която преобразува дадено число от двоична в десетична бройна система.
14. Напишете програма, която преобразува дадено число от десетична в шестнайсетична бройна система.
15. Напишете програма, която преобразува дадено число от шестнайсетична в десетична бройна система.
16. Напишете програма, която по дадено число N отпечатва числата от 1 до N, разбъркани в случаен ред.
17. Напишете програма, която за дадени две числа, намира най-големия им общ делител.
18. Напишете програма, която по дадено число n, извежда матрица във формата на спирала:

    пример при n=4

1	2	3	4
12	13	14	5
11	16	15	6
10	9	8	7

Решения и упътвания

1. Използвайте `for` цикъл.
2. Използвайте `for` цикъл и оператора `%` за намиране на остатък при целочислено деление. Едно число `num` не се дели на 3 и на 7 едновременно, ако (`num % (3*7) == 0`).
3. Първо прочетете броя числа, примерно в променлива `n`. След това въведете `n` числа последователно с един `for` цикъл. Докато въвеждате всяко следващо число запазвайте в две променливи най-малкото и най-голямото число до момента.
4. Номерирайте картите от 2 до 14 (тези числа ще съответстват на картите от 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A). Номерирайте боите от 1 до 4 (1 – спатия, 2 – каро, 3 – купа, 4 – пика). Сега вече можете да завъртите 2 вложени цикъла и да отпечатате всяка от картите.
5. Числата на Фибоначи започват от 0 и 1, като всяко следващо се получава като сума от предходните две. Можете да намерите първите

`n` числа на Фибоначи с **`for`** цикъл от 1 до **`n`**, като на всяка итерация пресмятате поредното число, използвайки предходните две (които ще пазите в две допълнителни променливи).

6. Умножете числата от K+1 до N.

7. Вариант за решение е поотделно да пресмятате всеки от факториелите и накрая да извършвате съответните операции с резултатите. Помислете как можете да оптимизирате пресмятанията, за да не смятате прекалено много факториели! При обикновени дроби, съставени от факториели има много възможности за съкращение на еднакви множители в числителя и знаменателя. Тези оптимизации не само ще намалят изчисленията и ще увеличат производителността, но ще ви избавят и от препълвания в някои ситуации.

8. Погледнете предходната задача.

9. Задачата може да решите с **`for`**-цикъл за **`k`**=0…**`n`**, като ползвате три допълнителни променливи **`factoriel`**, **`power`** и **`sum`**, в които да пазите за k-тата итерация на цикъла съответно k!, x^k и сумата на първите k члена на редицата. Ако реализацията ви е добра, Трябва да имате само един цикъл и не трябва да ползвате външни функции за изчисление на факториел и за степенуване.

10. Трябва да използвате два вложени цикъла, по подобие на задачата за отпечатване на триъгълник с числа. Външният цикъл трябва да върти от 1 до N, а вътрешният – от стойността на външния до стойността на външния + N - 1.

11. Броят на нулите в края на n! зависи от това, колко пъти числото 10 е делител на факториела. Понеже произведението 1*2*3…*n има повече на брой делители 2, отколкото 5, а 10 = 2 * 5, то броят нули в n! е точно толкова, колкото са множителите със стойност 5 в произведението 1*2*3….*n. Понеже всяко пето число се дели на 5, а всяко 25-то число се дели на 5 двукратно и т.н., то броя нули в n! е сумата: n/5 + n/25 + n/125 + …

12. Прочетете в Уикипедия какво представляват бройните системи: http://en.wikipedia.org/wiki/Numeral_system. След това помислете как можете да преминавате от десетична в друга бройна система. Помислете и за обратното – преминаване от друга бройна система към десетична. Ако се затрудните, вижте главата "Бройни системи".

13. Погледнете предходната задача.

14. Погледнете предходната задача.

15. Погледнете предходната задача.

16. Потърсете в Интернет информация за класа **`System.Random`**. Прочетете в Интернет за масиви (или в следващата глава). Направете масив с N елемента и запишете в него числата от 1 до N. След това достатъчно

много пъти (помислете точно колко) разменяйте двойки случайни числа от масива.

17. Потърсете в интернет за алгоритъма на Евклид.
18. Трябва да използвате двумерен масив. Потърсете в интернет или вижте главата "Масиви"

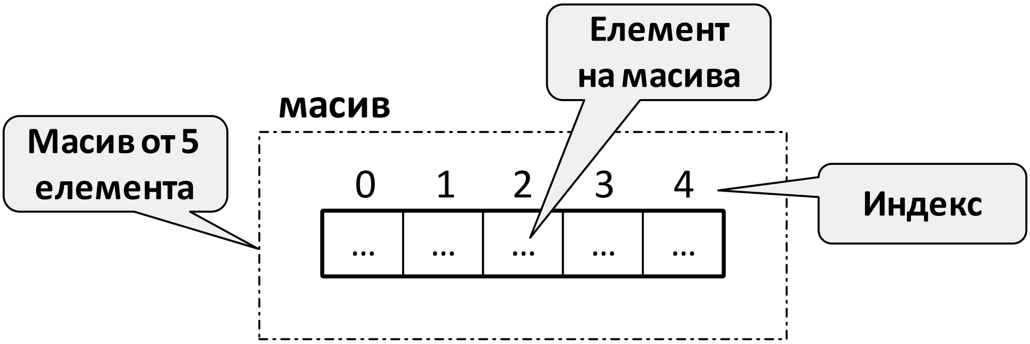
Глава 7. Масиви

В тази тема...

В настоящата тема ще се запознаем с масивите като средство за обработка на поредица от еднакви по тип елементи. Ще обясним какво представляват масивите, как можем да декларираме, създаваме, инициализираме и използваме масиви. Ще обърнем внимание на едномерните и многомерните масиви. Ще разгледаме различни начини за обхождане на масив, четене от стандартния вход и отпечатване на стандартния изход. Ще дадем много примери за задачи, които се решават с използването на масиви и ще демонстрираме колко полезни са те.

Какво е "масив"?

Масивите са неизменна част от повечето езици за програмиране. Те представляват съвкупности от променливи, които наричаме **елементи**:



Елементите на масивите в C# са номерирани с числата 0, 1, 2, ... N-1. Тези номера на елементи се наричат **индекси**. Броят елементи в даден масив се нарича **дължина на масива**.

Всички елементи на даден масив са от един и същи тип, независимо дали е **примитивен** или **референтен**. Това ни помага да представим група от еднородни елементи като подредена свързана последователност и да ги обработваме като едно цяло.

Масивите могат да бъдат от различни размерности, като най-често използвани са **едномерните** и **двумерните** масиви. Едномерните масиви се наричат още вектори, а двумерните – матрици.

Деклариране и заделяне на масиви

В C# масивите имат фиксирана дължина, която се указва при инициализирането им и определя броя на елементите им. След като веднъж е зададена дължината на масив при неговото създаване, след това не е възможно да се променя.

Деклариране на масив

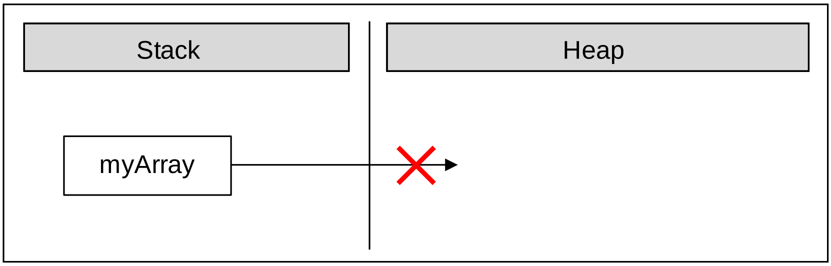
Масиви в C# декларираме по следния начин:

```
int[] myArray;
```

В примера променливата **`myArray`** е името на масива, който е от тип (**`int[]`**), т.е. декларирали сме масив от цели числа. С **`[]`** се обозначава, че променливата, която декларираме е масив от елементи, а не единичен елемент.

При декларация на променливата от тип масив, тя представлява референция (**reference**), която няма стойност (сочи към **`null`**), тъй като още не е заделена памет за елементите на масива.

Ето как изглежда една променлива от тип масив, която е декларирана, но още не е заделена памет за елементите на масива:



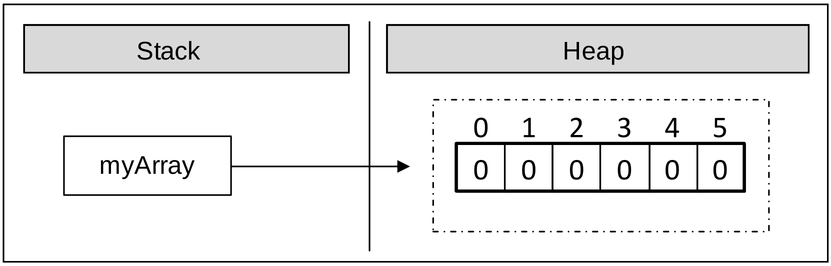
В стека за изпълнение на програмата се заделя променлива с име **`myArray`** и в нея се поставя стойност **`null`** (липса на стойност).

Създаване (заделяне) на масив – оператор new

В C# масив се създава с ключовата дума **`new`**, която служи за заделяне (алокиране) на памет:

```
int[] myArray = new int[6];
```

В примера заделяме масив с размер 6 елемента от целочисления тип **`int`**. Това означава, че в динамичната памет (heap) се заделя участък от 6 последователни цели числа, които се инициализират със стойност 0:



Картинката показва, че след заделянето на масива променливата **`myArray`** сочи към адрес в динамичната памет, където се намира нейната стойност. Елементите на масивите винаги се съхраняват в динамичната памет (в т. нар. **heap**).

При заделянето на масив в квадратните скоби се задава броят на елементите му (цяло неотрицателно число) и така се фиксира неговата дължина. Типът на елементите се пише след **`new`**, за да се укаже за какви точно елементи трябва да се задели памет.

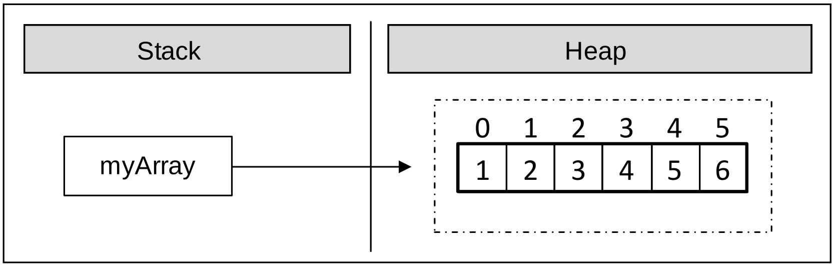
Инициализация на масив. Стойности по подразбиране

Преди да използваме елемент от даден масив той трябва да има начална стойност. В някои езици за програмиране не се задават начални стойности по подразбиране, и тогава при опит за достъпване на даден елемент може да възникне грешка. В C# всички променливи, включително и елементите на масивите имат начална стойност по подразбиране (default initial value). Тази стойност е равна на 0 при числените типове или неин еквивалент при нечислени типове (например **`null`** за референтни типове и **`false`** за булевия тип).

Разбира се, начални стойности можем да задавам и изрично. Това може да стане по различни начини. Ето един от тях:

```
int[] myArray = { 1, 2, 3, 4, 5, 6 };
```

В този случай създаваме и инициализираме елементите на масива едновременно. Ето как изглежда масивът в паметта, след като стойностите му са инициализирани още в момента на деклариране:



При този синтаксис къдравите скоби заместват оператора **`new`** и между тях са изброени началните стойности на масива, разделени със запетаи. Техния брой определя дължината му.

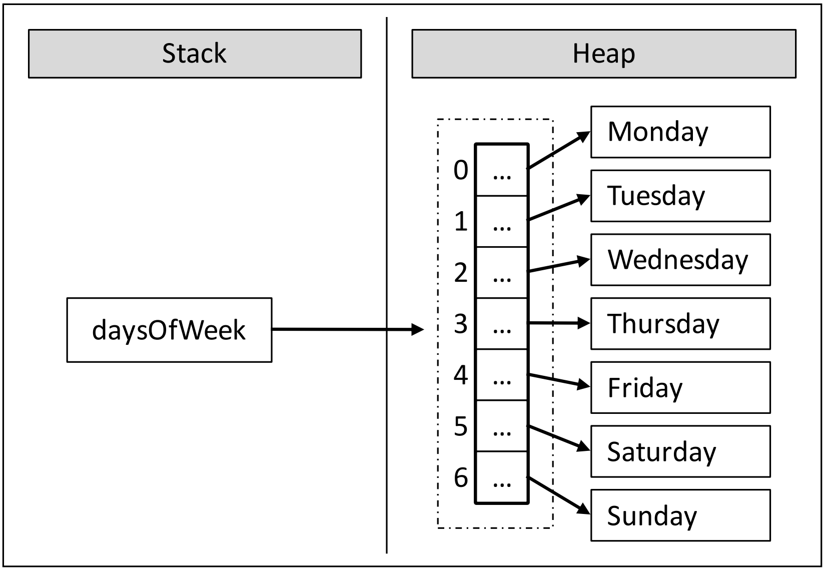
Деклариране и инициализиране на масив – пример

Ето още един пример за деклариране и непосредствено инициализиране на масив:

```
string[] daysOfWeek =
  { "Monday", "Tuesday", "Wednesday","Thursday", "Friday",
  "Saturday", "Sunday" };
```

В случая масивът се заделя със 7 елемента от тип **`string`**. Типът **`string`** е референтен тип (обект) и неговите стойности се пазят в динамичната памет. В стека се заделя променливата **`daysOfWeek`**, която сочи към участък в динамичната памет, който съдържа елементите на масива. Всеки от тези 7 елемента е обект от тип символен низ (**`string`**), който сам по себе си сочи към друга област от динамичната памет, в която се пази стойността му.

Ето как е разположен масивът в паметта:



Граници на масив

Масивите по подразбиране са **нулево-базирани**, т.е. номерацията на елементите започва от **0**. Първият елемент има индекс 0, вторият 1 и т.н. Ако един масив има **N** елемента, то последният елемент се намира на индекс **N-1**.

Достъп до елементите на масив

Достъпът до елементите на масивите е пряк и се осъществява по индекс. Всеки елемент може да се достъпи с името на масива и съответния му индекс (пореден номер), поставен в квадратни скоби. Можем да осъществим достъп до даден елемент както за четене така и за писане т.е. да го третираме като най-обикновена променлива.

Пример за достъп до елемент на масив:

```
myArray[index] = 100;
```

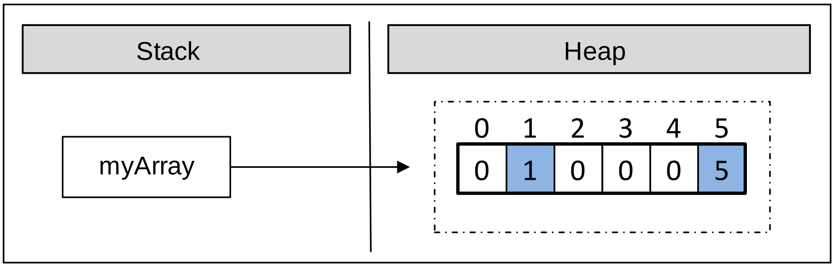
В горния пример присвояваме стойност 100 на елемента, намиращ се на позиция **`index`**.

Ето един пример, в който заделяме масив от числа и след това променяме някои от елементите му:

```
int[] myArray = new int[6];
```

```
myArray[1] = 1;
myArray[5] = 5;
```

След промяната на елементите, масивът се представя в паметта по следния начин:



Както се вижда, всички елементи, с изключение на тези, на които изрично сме задали стойност, са инициализирани с 0 при заделянето на масива.

Масивите могат да се обхождат с помощта на някоя от конструкциите за **цикъл**, като най-често използван е класическият **`for`** цикъл:

```
int[] arr = new int[5];
for (int i = 0; i < arr.Length; i++)
{
   arr[i] = i;
}
```

Излизане от границите на масив

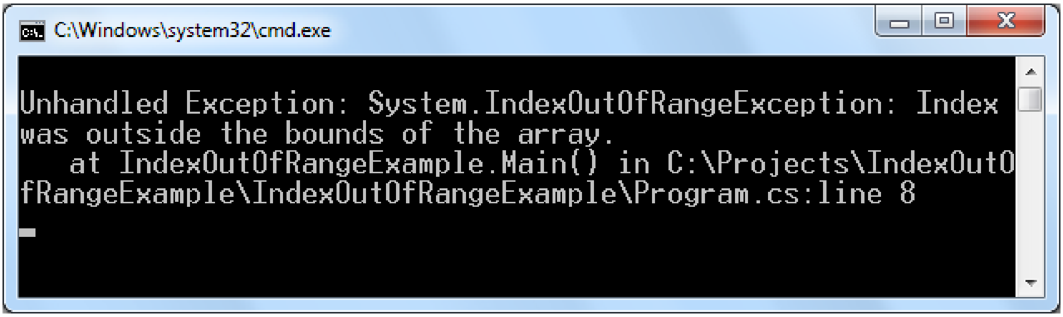
При всеки достъп до елемент на масив .NET Framework прави автоматична проверка, дали индексът е валиден или е извън границите на масива. При опит за достъп до невалиден елемент се хвърля изключение от тип **`System.IndexOutOfRangeException`**. Автоматичната проверка за излизане от границите на масива е изключително полезна за разработчиците и води до ранно откриване на грешки при работа с масиви. Естествено, тези проверки си имат и своята цена и тя е леко намаляване на производителността, която е нищожна в сравнение с избягването на грешки от тип "излизане от масив", "достъп до незаделена памет" и други.

Ето един пример, в който се опитваме да извлечем елемент, който се намира извън границите на масива:

IndexOutOfRangeExample.cs

```
class IndexOutOfRangeExample
{
   static void Main()
   {
      int[] myArray = { 1, 2, 3, 4, 5, 6 };
      Console.WriteLine(myArray[6]);
   }
}
```

В горния пример създаваме масив, който съдържа 6 цели числа. Първият елемент се намира на индекс 0, а последният – на индекс 5. Опитваме се да изведем на конзолата елемент, който се намира на индекс 6, но понеже такъв не съществува, това води до възникване на изключение:



```
Unhandled Exception: System.IndexOutOfRangeException: Index
was outside the bounds of the array.
   at IndexOutOfRangeExample.Main() in C:\Projects\IndexOutO
fRangeExample\IndexOutOfRangeExample\Program.cs:line 8
```

Обръщане на масив в обратен ред – пример

В следващия пример ще видим как може да променяме елементите на даден масив като ги достъпваме по индекс. Целта на задачата е да се подредят в обратен ред (отзад напред) елементите на даден масив. Ще обърнем елементите на масива, като използваме помощен масив, в който да запазим елементите на първия, но в обратен ред. Забележете, че дължината на масивите е еднаква и остава непроменена след първоначалното им заделяне:

ArrayReverseExample.cs

```
class ArrayReverseExample
{
   static void Main()
   {
      int[] array = { 1, 2, 3, 4, 5 };
      // Get array size
      int length = array.Length;
      // Declare and create the reversed array
      int[] reversed = new int[length];

      // Initialize the reversed array
      for (int index = 0; index < length; index++)
      {
         reversed[length - index - 1] = array[index];
      }

      // Print the reversed array
      for (int index = 0; index < length; index++)
      {
        Console.Write(reversed[index] + " ");
      }
    }
}
// Output: 5 4 3 2 1
```

Примерът работи по следния начин: първоначално създаваме едномерен масив от тип **int** и го инициализираме с цифрите от 1 до 5. След това запазваме дължината на масива в целочислената променлива **length**. Забележете, че се използва свойството **Length**, което връща броя на елементите на масива. В C# всеки масив знае своята дължина.

След това декларираме масив **reversed** със същия размер **length**, в който ще си пазим елементите на оригиналния масив, но в обратен ред.

За да извършим обръщането на елементите използваме цикъл **for**, като на всяка итерация увеличаваме водещата променлива **index** с единица и така си осигуряваме последователен достъп до всеки елемент на масива **array**. Критерият за край на цикъла ни подсигурява, че масивът ще бъде обходен от край до край.

Нека проследим последователно какво се случва при итериране върху масива **array**. При първата итерация на цикъла, **index** има стойност 0. С **array[index]** достъпваме първия елемент на масива **array**, а съответно с **reversed[length - index - 1]** достъпваме последния елемент на новия масив **reversed** и извършваме присвояване. Така на последния елемент на **reversed** присвоихме първия елемент на **array**. На всяка следваща итерация **index** се увеличава с единица, позицията в **array** се увеличава с единица, а в **reversed** се намаля с единица.

В резултат обърнахме масива в обратен ред и го отпечатахме. В примера показахме последователно обхождане на масив, което може да се извърши и с другите видове цикли (например **while** и **foreach**).

Четене на масив от конзолата

Нека разгледаме как можем да прочетем стойностите на масив от конзолата. Ще използваме **for** цикъл и средствата на .NET Framework за четене от конзолата.

Първоначално, прочитаме един ред от конзолата с помощта на **Console.ReadLine()**, след това преобразуваме прочетения ред към цяло число с помощта на **int.Parse()** и го присвояваме на **n**. Числото **n** ползваме по-нататък като размер на масива.

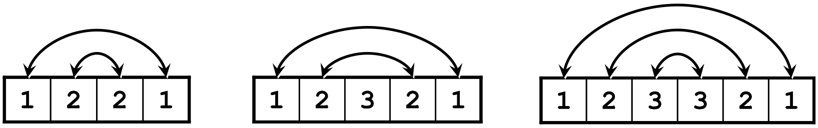
```
int n = int.Parse(Console.ReadLine());
int[] array = new int[n];
```

Отново използваме цикъл, за да обходим масива. На всяка итерация присвояваме на текущия елемент прочетеното от конзолата число. Цикълът ще се завърти **n** пъти т.е. ще обходи целия масив и така ще прочетем стойност за всеки един елемент от масива:

```
for (int i = 0; i < n; i++)
{
  array[i] = int.Parse(Console.ReadLine());
}
```

Проверка за симетрия на масив – пример

Един масив е симетричен, ако първият и последният му елемент са еднакви и същевременно вторият и предпоследният му елемент също са еднакви и т.н. На картинката са дадени няколко примера за симетрични масиви:



В следващия примерен код ще видим как можем да проверим дали даден масив е симетричен:

```
Console.Write("Enter a positive integer: ");
int n = int.Parse(Console.ReadLine());
int[] array = new int[n];

Console.WriteLine("Enter the values of the array:");

for (int i = 0; i < n; i++)
{
  array[i] = int.Parse(Console.ReadLine());
}

bool symmetric = true;
for (int i = 0; i < array.Length / 2; i++)
{
  if (array[i] != array[n - i - 1])
  {
    symmetric = false;
  }
}

Console.WriteLine("Is symmetric? {0}", symmetric);
```

Тук отново създаваме масив и прочитаме елементите му от конзолата. За да проверим дали масивът е симетричен, трябва да го обходим само до средата. Тя е елементът с индекс **`array.Length / 2`**. При нечетна дължина на масива този индекс е средният елемент, а при нечетна – елементът вляво от средата (понеже средата е между два елемента).

За да определим дали даденият масив е симетричен ще ползваме **булева** променлива, като първоначално приемаме, че масивът е симетричен.

Обхождаме масива и сравняваме първия с последния елемент, втория с предпоследния и т.н. Ако за някоя итерация се окаже, че стойностите на сравняваните елементи не съвпадат, булевата променлива получава стойност **`false`**, т.е. масивът не е симетричен.

Накрая извеждаме на конзолата стойността на булевата променлива.

Отпечатване на масив на конзолата

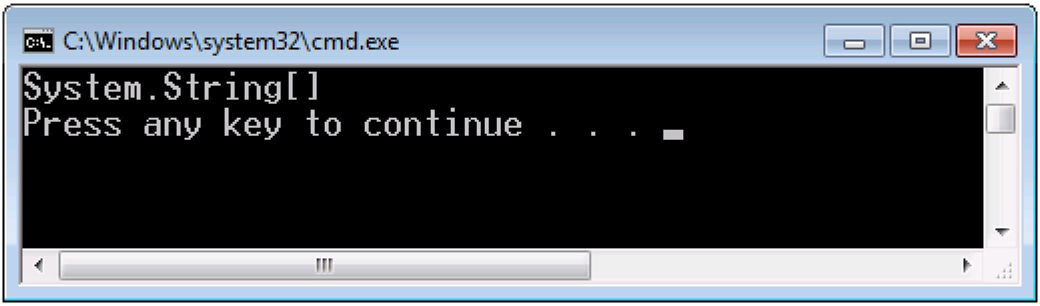
Често се налага след като сме обработвали даден масив да изведем елементите му на конзолата, било то за тестови или други цели.

Отпечатването на елементите на масив става по подобен начин на инициализирането на елементите му, а именно като използваме цикъл, който обхожда масива. Няма строги правила за начина на извеждане на елементите, но често пъти се ползва подходящо форматиране.

Често срещана грешка е опит да се изведе на конзолата масив директно, по следния начин:

```
string[] array = { "one", "two", "three", "four" };
Console.WriteLine(array);
```

Този код за съжаление не отпечатва съдържанието на масива, а неговия тип. Ето как изглежда резултатът от изпълнението на горния пример:



```
System.String[]
Press any key to continue . . .
```

За да изведем коректно елементите на масив на конзолата можем да използваме **`for`** цикъл:

```
string[] array = { "one", "two", "three", "four" };

for (int index = 0; index < array.Length; index++)
{
   // Print each element on a separate line
   Console.WriteLine("Element[{0}] = {1}", index, array[index]);
}
```

Обхождаме масива с цикъл **`for`**, който извършва **`array.Length`** на брой итерации, и с помощта на метода **`Console.WriteLine()`** извеждаме поредния му елемент на конзолата чрез форматиращ стринг. Резултатът е следният:

```
Element[0] = one
Element[1] = two
Element[2] = three
Element[3] = four
```

Итерация по елементите на масив

Както разбрахме до момента, итерирането по елементите на масив е една от основните операции при обработката на масиви. Итерирайки последователно по даден масив можем да достъпим всеки елемент с помощта на индекс и да го обработваме по желан от нас начин. Това може да стане с всички видове конструкции за цикъл, които разгледахме в предната тема, но най-подходящ за това е стандартният **`for`** цикъл. Нека разгледаме как точно става обхождането на масиви.

Итерация с for цикъл

Добра практика е да използваме **`for`** цикъл при работа с масиви и изобщо при индексирани структури. Ето един пример, в който удвояваме стойността на всички елементи от даден масив с числа и го принтираме:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };

Console.Write("Output: ");
for (int index = 0; index < array.Length; index++)
{
   // Doubling the number
   array[index] = 2 * array[index];
   // Print the number
   Console.Write(array[index] + " ");
}
// Output: 2 4 6 8 10
```

Чрез **`for`** цикъла можем да имаме постоянен поглед върху текущия индекс на масива и да достъпваме точно тези елементи, от които имаме нужда. Итерирането може да не се извършва последователно т.е. индексът, който **`for`** цикъла ползва може да прескача по елементите според нуждите на нашия алгоритъм. Например можем да обходим част от даден масив, а не всичките му елементи:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };

Console.Write("Output: ");
for (int index = 0; index < array.Length; index += 2)
{
   array[index] = array[index] * array[index];
   Console.Write(array[index] + " ");
}
// Output: 1 9 25
```

В горния пример обхождаме всички елементи на масива, намиращи се на четни позиции и повдигаме на квадрат стойността във всеки от тях.

Понякога е полезно да обходим масив отзад напред. Можем да постигнем това по напълно аналогичен начин, с разликата, че **for** цикълът ще започва с начален индекс, равен на индекса на последния елемент на масива, и ще се намаля на всяка итерация докато достигне 0 (включително). Ето един такъв пример:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };

Console.Write("Reversed: ");
for (int index = array.Length - 1; index >= 0; index--)
{
   Console.Write(array[index] + " ");
}
// Reversed: 5 4 3 2 1
```

В горния пример обхождаме масива отзад напред последователно и извеждаме всеки негов елемент на конзолата.

Итерация с цикъл foreach

Една често използвана конструкция за итерация по елементите на масив е така нареченият **foreach**. Конструкцията на **foreach** цикъла в C# е следната:

```
foreach (var item in collection)
{
   // Process the value here
}
```

При тази конструкция **var** е типът на елементите, които обхождаме т.е. типа на масива, **collection** е масивът (или някаква друга колекция от елементи), а **item** е текущият елемент от масива на всяка една стъпка от обхождането.

Цикълът **foreach** притежава в голяма степен свойствата на **for** цикъла. Отличава се с това, че преминаването през елементите на масива (или на колекцията, която се обхожда), се извършва винаги от край до край. При него не е достъпен индексът на текущата позиция, а просто се обхождат всички елементи в ред, определен от самата колекция, която се обхожда. За масивите редът на обхождане е последователно от нулевия към последния елемент.

Този цикъл се използва, когато нямаме нужда да променяме елементите на масива, а само да ги четем и да обхождаме целия масив.

Итерация с цикъл foreach – пример

В следващия пример ще видим как да използваме конструкцията на **`foreach`** цикъла за обхождане на масиви:

```
string[] capitals =
  { "Sofia", "Washington", "London", "Paris" };

foreach (string capital in capitals)
{
  Console.WriteLine(capital);
}
```

След като сме си декларирали масив от низове **`capitals`**, с **`foreach`** го обхождаме и извеждаме елементите му на конзолата. Текущият елемент на всяка една стъпка се пази в променливата **`capital`**. Ето какъв резултат се получава при изпълнението на примера:

```
Sofia
Washington
London
Paris
```

Многомерни масиви

До момента разгледахме работата с едномерните масиви, известни в математиката като "вектори". В практиката, обаче, често се ползват масиви с повече от едно измерения. Например стандартна шахматна дъска се представя лесно с двумерен масив с размер 8 на 8 (8 полета в хоризонтална посока и 8 полета във вертикална посока).

Какво е "многомерен масив"? Какво е "матрица"?

Всеки допустим в C# тип може да бъде използван за тип на елементите на масив. Масивите също може да се разглеждат като допустим тип. Така можем да имаме масив от масиви, който ще разгледаме по-нататък.

Едномерен масив от цели числа декларираме с **`int[]`**, а двумерен масив с **`int[,]`**. Следния пример показва това:

```
int[,] twoDimentionalArray;
```

Такива масиви ще наричаме **двумерни**, защото имат две измерения или още **матрици** (терминът идва от математиката). Масиви с повече от едно измерение ще наричаме **многомерни**.

Аналогично можем да декларираме и **тримерни** масиви като добавим още едно измерение:

```
int[,,] threeDimentionalArray;
```

На теория няма ограничения за броя на размерностите на тип на масив, но в практиката масиви с повече от две размерности са рядко използвани и затова ще се спрем по-подробно на двумерните масиви.

Деклариране и заделяне на многомерен масив

Многомерните масиви се декларират по начин аналогичен на едномерните. Всяка тяхна размерност (освен първата) означаваме със запетая:

```
int[,] intMatrix;
float[,] floatMatrix;
string[,,] strCube;
```

Горният пример показва как да създадем двумерни и тримерни масиви. Всяка размерност в допълнение на първата отговаря на една запетая в квадратните скоби `[]`.

Памет за многомерни размери се заделя като се използва ключовата дума `new` и за всяка размерност в квадратни скоби се задава размерът, който е необходим:

```
int[,] intMatrix = new int[3, 4];
float[,] floatMatrix = new float[8, 2];
string[,,] stringCube = new string[5, 5, 5];
```

В горния пример `intMatrix` е двумерен масив с 3 елемента от тип `int[]` и всеки от тези 3 елемента има размерност 4. Така представени, двумерните масиви изглеждат трудни за осмисляне. Затова може да ги разглеждаме като двумерни **матрици**, които имат редове и колони за размерности:

	0	1	2	3
0	1	3	6	2
1	8	5	9	1
2	4	7	3	0

Редовете и колоните на квадратните матрици се номерират с индекси от 0 до n-1. Ако един двумерен масив има размер `m` на `n`, той има точно `m*n` елемента.

Инициализация на двумерен масив

Инициализацията на многомерни масиви е аналогична на инициализацията на едномерните. Стойностите на елементите могат да се изброяват непосредствено след декларацията:

```
int[,] matrix =
{
   {1, 2, 3, 4}, // row 0 values
   {5, 6, 7, 8}, // row 1 values
};
// The matrix size is 2 x 4 (2 rows, 4 cols)
```

В горния пример инициализираме двумерен масив с цели числа с 2 реда и 4 колони. Във външните фигурни скоби се поставят елементите от първата размерност, т.е. редовете на двумерната матрица. Всеки ред представлява едномерен масив, който се инициализира по познат за нас начин.

Достъп до елементите на многомерен масив

Матриците имат две размерности и съответно всеки техен елемент се достъпва с помощта на два индекса – един за редовете и един за колоните. Многомерните масиви имат различен индекс за всяка размерност.

Всяка размерност в многомерен масив започва от индекс нула.

Нека разгледаме следния пример:

```
int[,] matrix =
{
   {1, 2, 3, 4},
   {5, 6, 7, 8},
};
```

Масивът **`matrix`** има 8 елемента, разположени в 2 реда и 4 колони. Всеки елемент може да се достъпи по следния начин:

```
matrix[0, 0]  matrix[0, 1]  matrix[0, 2]  matrix[0, 3]
matrix[1, 0]  matrix[1, 1]  matrix[1, 2]  matrix[1, 3]
```

В горния пример виждаме как да достъпим всеки елемент по индекс. Ако означим индекса по редове с **`row`**, а индекса по колони с **`col`**, тогава достъпа до елемент от двумерен масив има следния общ вид:

```
matrix[row, col]
```

При многомерните масиви всеки елемент се идентифицира уникално с толкова на брой индекси, колкото е размерността на масива:

```
nDimensionalArray[index1, … , indexN]
```

Дължина на многомерен масив

Всяка размерност на многомерен масив има собствена дължина, която е достъпна по време на изпълнение на програмата. Нека разгледаме следния пример за двумерен масив:

```
int[,] matrix =
{
   {1, 2, 3, 4},
   {5, 6, 7, 8},
};
```

Можем да извлечем броя на редовете на този двумерен масив чрез **matrix.GetLength(0)**, а дължината на всеки от редовете (т.е. броя колони) с **matrix.GetLength(1)**.

Отпечатване на матрица – пример

Чрез следващия пример ще демонстрираме как можем да отпечатваме двумерни масиви на конзолата:

```
// Declare and initialize a matrix of size 2 x 4
int[,] matrix =
{
   {1, 2, 3, 4}, // row 0 values
   {5, 6, 7, 8}, // row 1 values
};

for (int row = 0; row < matrix.GetLength(0); row++)
{
   for (int col = 0; col < matrix.GetLength(1); col++)
   {
      Console.Write(matrix[row, col]);
   }
   Console.WriteLine();
}
// Print the matrix on the console
```

Първо декларираме и инициализираме масива, който искаме да обходим и да отпечатаме на конзолата. Масивът е двумерен и затова използваме един цикъл, който ще се движи по редовете и втори, вложен цикъл, който за всеки ред ще се движи по колоните на масива. За всяка итерация по подходящ начин извеждаме текущия елемент на масива като го достъпваме по неговите два индекса (ред и колона). В крайна сметка, ако изпълним горния програмен фрагмент, ще получим следния резултат:

```
1 2 3 4
5 6 7 8
```

Четене на матрица от конзолата – пример

Нека видим как можем да прочетем двумерен масив (матрица) от конзолата. Това става като първо въведем големините на двете размерности, а след това с два вложени цикъла въвеждаме всеки от елементите му:

```
Console.Write("Enter the number of the rows: ");
int rows = int.Parse(Console.ReadLine());

Console.Write("Enter the number of the columns: ");
int cols = int.Parse(Console.ReadLine());

int[,] matrix = new int[rows, cols];

Console.WriteLine("Enter the cells of the matrix:");

for (int row = 0; row < rows; row++)
{
   for (int col = 0; col < cols; col++)
   {
      Console.Write("matrix[{0},{1}] = ",row, col);
      matrix[row, col] = int.Parse(Console.ReadLine());
   }
}

for (int row = 0; row < matrix.GetLength(0); row++)
{
   for (int col = 0; col < matrix.GetLength(1); col++)
   {
      Console.Write(" " + matrix[row, col]);
   }
   Console.WriteLine();
}
```

Ето как може да изглежда програмата в действие (в случая въвеждаме масив с размер 3 реда на 2 колони):

```
Enter the number of the rows: 3
Enter the number of the columns: 2
Enter the cells of the matrix:
matrix[0,0] = 2
matrix[0,1] = 3
matrix[1,0] = 5
matrix[1,1] = 10
matrix[2,0] = 8
matrix[2,1] = 9
```

```
2 3
5 10
8 9
```

Максимална площадка в матрица – пример

В следващия пример ще решим една по-интересна задача: дадена е правоъгълна матрица с числа и трябва да намерим в нея максималната подматрица с размер 2 x 2 и да я отпечатаме на конзолата. Под максимална подматрица ще разбираме подматрица, която има максимална сума на елементите, които я съставят. Ето едно примерно решение на задачата:

MaxPlatform2x2.cs

```
class MaxPlatform2x2
{
  static void Main()
  {
    // Declare and initialize the matrix
    int[,] matrix = {
       { 0, 2, 4, 0, 9, 5 },
       { 7, 1, 3, 3, 2, 1 },
       { 1, 3, 9, 8, 5, 6 },
       { 4, 6, 7, 9, 1, 0 }
    };

    // Find the maximal sum platform of size 2 x 2
    int bestSum = int.MinValue;
    int bestRow = 0;
    int bestCol = 0;

    for (int row = 0; row < matrix.GetLength(0) - 1; row++)
    {
      for (int col = 0; col < matrix.GetLength(1) - 1; col++)
      {
        int sum = matrix[row, col] + matrix[row, col + 1] +
          matrix[row + 1, col] + matrix[row + 1, col + 1];
        if (sum > bestSum)
        {
          bestSum = sum;
          bestRow = row;
          bestCol = col;
        }
      }
    }

    // Print the result
    Console.WriteLine("The best platform is:");
    Console.WriteLine("  {0} {1}",
          matrix[bestRow, bestCol],
          matrix[bestRow, bestCol + 1]);
        Console.WriteLine("  {0} {1}",
          matrix[bestRow + 1, bestCol],
          matrix[bestRow + 1, bestCol + 1]);
        Console.WriteLine("The maximal sum is: {0}", bestSum);
    }
}
```

Ако изпълним програмата, ще се убедим, че работи коректно:

```
The best platform is:
  9 8
  7 9
The maximal sum is: 33
```

Нека сега обясним реализирания алгоритъм. В началото на програмата си създаваме двумерен масив, състоящ се от цели числа. Декларираме помощни променливи **`bestSum`**, **`bestRow`**, **`bestCol`** и инициализираме **`bestSum`** с минималната за типа **`int`** стойност (така че всяка друга да е по-голяма от нея).

В променливата **`bestSum`** ще пазим текущата максимална сума, а в **`bestRow`** и **`bestCol`** ще пазим най-добрата до момента подматрица, т.е. текущият ред и колона, които са начало на подматрицата с размери 2 х 2, имаща сума на елементите **`bestSum`**.

За да достъпим всички елементи на подматрица 2 х 2 са ни необходими индексите на първия ѝ елемент. Като ги имаме лесно можем да достъпим другите 3 елемента по следния начин:

```
matrix[row, col]
matrix[row, col + 1]
matrix[row + 1, col]
matrix[row + 1, col + 1]
```

В горния пример **`row`** и **`col`** са индексите на отговарящи на първия елемент на матрица с размер 2 х 2, която е част от матрицата **`matrix`**.

След като вече разбрахме как да достъпим четирите елемента на матрица с размер 2 х 2, започващи от даден ред и колона, можем да разгледаме алгоритъма, по който ще намерим максималната такава матрица 2 х 2.

Трябва да обходим всеки елемент от главната матрица до предпоследния ред и предпоследната колона. Това правим с два вложени цикъла по променливите **`row`** и **`col`**. Забележете, че не обхождаме матрицата от край до край, защото при опит да достъпим индекси **`row + 1`** или **`col + 1`** ще излезем извън границите на масива ако сме на последния ред или колона и ще възникне изключение **`System.IndexOutOfRangeException`**.

Достъпваме съседните елементи на всеки текущ начален елемент на подматрица с размер 2 x 2 и ги сумираме. След това проверяваме дали текущата ни сума е по-голяма от текущата най-голяма сума. Ако е така текущата сума става текуща най-голяма сума и текущите индекси стават **`bestRow`** и **`bestCol`**. Така след пълното обхождане на главната матрица ще намерим максималната сума и индексите на началния елемент на подматрицата с размери 2 x 2, имаща тази най-голяма сума. Ако има няколко подматрици с еднаква максимална сума, ще намерим тази, която се намира на минимален ред и минимална колона в този ред.

В края на примера извеждаме на конзолата по подходящ начин търсената подматрица и нейната сума.

Масиви от масиви

В C# можем да използваме масив от масиви или така наречените **назъбени** (jagged) масиви.

Назъбените масиви представляват масиви от масиви или по-точно масиви, в които един ред на практика е също масив и може да има различна дължина от останалите в назъбения масив.

Деклариране и заделяне на масив от масиви

Единственото по-особено при назъбените масиви е, че нямаме само една двойка скоби, както при обикновените масиви, а имаме вече по една двойка скоби за всяко от измеренията. Заделянето става по същия начин:

```
int[][] jaggedArray;
jaggedArray = new int[2][];
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[3];
```

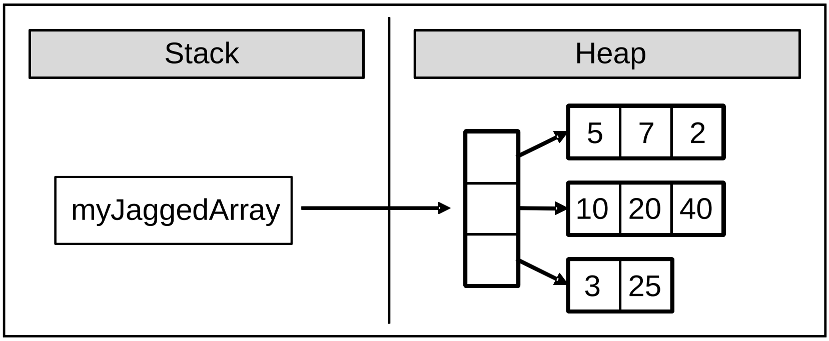
Ето как декларираме, заделяме и инициализираме един масив от масиви:

```
int[][] myJaggedArray = {
   new int[] {5, 7, 2},
   new int[] {10, 20, 40},
   new int[] {3, 25}
};
```

Разположение в паметта

На долната картинка може да се види вече дефинираният назъбен масив **`myJaggedArray`** или по-точно неговото разположение в паметта. Както се вижда самият назъбен масив представлява съвкупност от референции, а не съдържа самите масиви. Не се знае каква е размерността на масивите и затова CLR пази само референциите (указателите) към тях. След като заделим памет за някой от масивите-елементи на назъбения, тогава се насочва указателят към новосъздадения блок в динамичната памет.

Променливата **`myJaggedArray`** стои в стека за изпълнение на програмата и сочи към блок от динамичната памет, съдържащ поредица от три указатели към други блокове от паметта, всеки от които съдържа масив от цели числа – елементите на назъбения масив:



Инициализиране и достъп до елементите

Достъпът до елементите на масивите, които са част от назъбения, отново се извършва по индекса им. Ето пример за достъп до елемента с индекс 3 от масива, който се намира на индекс 0 в по-горе дефинирания назъбен масив **`jaggedArray`**:

```
myJaggedArray[0][3] = 45;
```

Елементите на назъбения масив може да са както едномерни масиви, така и многомерни такива. Ето един пример за назъбен масив от двумерни масиви:

```
int[][,] jaggedOfMulti = new int[2][,];
jaggedOfMulti[0] = new int[,] { { 5, 15 }, { 125, 206 } };
jaggedOfMulti[1] = new int[,] { { 3, 4, 5 }, { 7, 8, 9 } };
```

Триъгълник на Паскал – пример

В следващия пример ще използваме назъбен масив, за да генерираме и визуализираме триъгълника на Паскал. Както знаем от математиката, първият ред на триъгълника на Паскал съдържа числото 1, а всяко число от всеки следващ ред се образува като се съберат двете числа от горния ред над него. Триъгълникът на Паскал изглежда по следния начин:

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
       . . .
```

За да получим триъгълника на Паскал до дадена височина, например 12, можем да заделим назъбен масив **`triangle[][]`**, който съдържа 1 елемент на нулевия си ред, 2 – на първия, 3 – на втория и т.н. Първоначално инициализираме **`triangle[0][0] = 1`**, а всички останали клетки на масива получават по подразбиране стойност **0** при заделянето им. След това въртим цикъл по редовете, в който от стойностите на ред **`row`** получаваме стойностите на ред **`row+1`**. Това става с вложен цикъл по колоните на текущия ред, следвайки дефиницията за стойностите в триъгълника на Паскал: прибавяме стойността на текущата клетка от текущия ред (**`triangle[row][col]`**) към клетката под нея (**`triangle[row+1][col]`**) и клетката под нея вдясно (**`triangle[row+1][col+1]`**). При отпечатването се добавят подходящ брой интервали отляво (чрез метода **`PadRight()`** на класа **`String`**), за да изглежда резултатът по-подреден.

Следва примерна реализация на описания алгоритъм:

`PascalTriangle.cs`

```
class PascalTriangle
{
  static void Main()
  {
    const int HEIGHT = 12;

    // Allocate the array in a triangle form
    long[][] triangle = new long[HEIGHT + 1][];

    for (int row = 0; row < HEIGHT; row++)
    {
      triangle[row] = new long[row + 1];
    }

    // Calculate the Pascal's triangle
    triangle[0][0] = 1;
    for (int row = 0; row < HEIGHT - 1; row++)
    {
      for (int col = 0; col <= row; col++)
      {
        triangle[row + 1][col] += triangle[row][col];
        triangle[row + 1][col + 1] += triangle[row][col];
      }
    }

    // Print the Pascal's triangle
    for (int row = 0; row < HEIGHT; row++)
    {
      Console.Write("".PadLeft((HEIGHT - row) * 2));
      for (int col = 0; col <= row; col++)
      {
        Console.Write("{0,3} ", triangle[row][col]);
      }
      Console.WriteLine();
    }
  }
}
```

Ако изпълним програмата, ще се убедим, че тя работи коректно и генерира триъгълника на Паскал със зададената височина:

```
                      1
                    1   1
                  1   2   1
                1   3   3   1
              1   4   6   4   1
            1   5  10  10   5   1
          1   6  15  20  15   6   1
        1   7  21  35  35  21   7   1
      1   8  28  56  70  56  28   8   1
    1   9  36  84 126 126  84  36   9   1
  1  10  45 120 210 252 210 120  45  10   1
1  11  55 165 330 462 462 330 165  55  11   1
```

Упражнения

1. Да се напише програма, която създава масив с 20 елемента от целочислен тип и инициализира всеки от елементите със стойност равна на индекса на елемента умножен по 5. Елементите на масива да се изведат на конзолата.
2. Да се напише програма, която чете два масива от конзолата и проверява дали са еднакви.
3. Да се напише програма, която сравнява два масива от тип **`char`** лексикографски (буква по буква) и проверява кой от двата е по-рано в лексикографската подредба.
4. Напишете програма, която намира максимална редица от последователни еднакви елементи в масив. Пример: {2, 1, 1, 2, 3, 3, **2, 2, 2**, 1} → {2, 2, 2}.
5. Напишете програма, която намира максималната редица от последователни нарастващи елементи в масив. Пример: {3, **2, 3, 4**, 2, 2, 4} → {2, 3, 4}.
6. Напишете програма, която намира максималната подредица от нарастващи елементи в масив **`arr[n]`**. Елементите може и да не са последователни. Пример: {9, 6, **2**, 7, **4**, 7, **6**, 5, **8**, 4} → {2, 4, 6, 8}.
7. Да се напише програма, която чете от конзолата две цели числа N и K (K<N), и масив от N елемента. Да се намерят тези K поредни елемента, които имат максимална сума.

8. Сортиране на масив означава да подредим елементите му в нарастващ (намаляващ) ред. Напишете програма, която сортира масив. Да се използва алгоритъма "`Selection sort`".

9. Напишете програма, която намира последователност от числа, чиито сума е максимална. Пример: {2, 3, -6, -1, **2, -1, 6, 4**, -8, 8} → **11**

10. Напишете програма, която намира най-често срещания елемент в масив. Пример: {**4**, 1, 1, **4**, 2, 3, **4**, **4**, 1, 2, **4**, 9, 3} → 4 (среща се 5 пъти).

11. Да се напише програма, която намира последователност от числа в масив, които имат сума равна на число, въведено от конзолата (ако има такава). Пример: {4, 3, 1, **4, 2, 5**, 8}, S=11 → {4, 2, 5}.

12. Напишете програма, която създава следните квадратни матрици и ги извежда на конзолата във форматиран вид. Размерът на матриците се въвежда от конзолата. Пример за `(4,4)`:

a)

1	**5**	**9**	**13**
2	**6**	**10**	**14**
3	**7**	**11**	**15**
4	**8**	**12**	**16**

b)

1	**8**	**9**	**16**
2	**7**	**10**	**15**
3	**6**	**11**	**14**
4	**5**	**12**	**13**

c)

7	**11**	**14**	**16**
4	**8**	**12**	**15**
2	**5**	**9**	**13**
1	**3**	**6**	**10**

d)*

1	**12**	**11**	**10**
2	**13**	**16**	**9**
3	**14**	**15**	**8**
4	**5**	**6**	**7**

13. Да се напише програма, която създава правоъгълна матрица с размер `n` на `m`. Размерността и елементите на матрицата да се четат от конзолата. Да се намери подматрицата с размер `(3,3)`, която има максимална сума.

14. Да се напише програма, която намира най-дългата последователност от еднакви `string` елементи в матрица. Последователност в матрица дефинираме като елементите са на съседни и са на същия ред,колона или диагонал.

ha	fifi	ho	hi
fo	**ha**	hi	xx
xxx	ho	**ha**	xx

→ ha, ha, ha

15. Да се напише програма, която създава масив с всички букви от латинската азбука. Да се даде възможност на потребител да въвежда дума от конзолата и в резултат да се извеждат индексите на буквите от думата.

16. Да се реализира двоично търсене (`binary search`) в **сортиран** целочислен масив.

17. Напишете програма, която сортира целочислен масив по алгоритъма "`merge sort`".

18. Напишете програма, която сортира целочислен масив по алгоритъма "`quick sort`".

19. Напишете програма, която намира всички прости числа в диапазона [1…10 000 000].

20. Напишете програма, която по дадени `N` числа и число `S`, проверявадали може да се получи сума равна на `S` с използване на подмасив от `N`-те числа (не непременно последователни).

    Пример: {2, **1, 2**, 4, 3, **5**, 2, **6**}, S = 14 → yes (1 + 2 + 5 + 6 = 14)

21. Напишете програма, която по дадени `N`, `K` и `S`, намира `K` на брой елементи измежду `N`-те числа, чиито сума е точно `S` или показва, че това е невъзможно.

22. Напишете програма, която прочита от конзолата масив от цели числа и премахва минимален на брой числа, така че останали числа да са сортирани в нарастващ ред. Отпечатайте резултата.

    Пример: {6, **1,** 4, **3**, 0, **3**, 6, **4**, **5**} → {1, 3, 3, 4, 5}

23. Напишете програма, която прочита цяло число `N` от конзолата и отпечатва всички пермутации на числата `[1…N]`.

    Пример: N = 3 → {1, 2, 3}, {1, 3, 2}, {2, 1, 3}, {2, 3, 1}, {3, 1, 2}, {3, 2, 1}

24. Напишете програма, която прочита цели числа `N` и `K` от конзолата и отпечатва всички вариации от `K` елемента на числата `[1…N]`.

    Пример: N = 3, K = 2 → {1, 1}, {1, 2}, {1, 3}, {2, 1}, {2, 2}, {2, 3}, {3, 1}, {3, 2}, {3, 3}

25. Напишете програма, която прочита цяло число `N` от конзолата и отпечатва всички комбинации от К елемента на числата `[1…N]`.

    Пример: N = 5, K = 2 → {1, 2}, {1, 3}, {1, 4}, {1, 5}, {2, 1}, {2, 3}, {2, 4}, {2, 5}, {3, 1}, {3, 4}, {3, 5}, {4, 5}

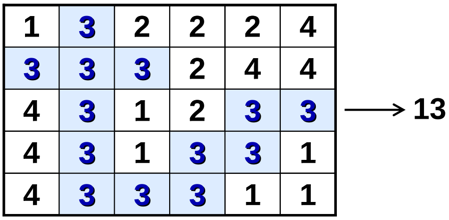
26. Напишете програма, която обхожда матрица (`NxN`) по следния начин:

    Пример за N=4:

16	15	13	10
14	12	9	6
11	8	5	3
7	4	2	1

7	11	14	16
4	8	12	15
2	5	9	13
1	3	6	10

27. *Напишете програма, която по подадена матрица намира най-голямата област от еднакви числа. Под област разбираме съвкупност от съседни (по ред и колона) елементи. Ето един пример, в който имаме област, съставена от 13 на брой еднакви елементи със стойност 3:



1	3	2	2	2	4
3	3	3	2	4	4
4	3	1	2	3	3
4	3	1	3	3	1
4	3	3	3	1	1

Решения и упътвания

1. Използвайте масив **`int[]`** и **`for`** цикъл.

2. Два масива са еднакви, когато имат еднаква дължина и стойностите на елементите в тях съответно съвпадат. Второто условие можете да проверите с **`for`** цикъл.

3. При лексикографската наредба символите се сравняват един по един като се започва от най-левия. При несъвпадащи символи по-рано е масивът, чийто текущ символ е по-рано в азбуката. При съвпадение се продължава със следващия символ вдясно. Ако се стигне до края на единия масив, по-краткият е лексикографски по-рано. Ако всички съответни символи от двата масива съвпаднат, то масивите са еднакви и никой о тях не е по-рано в лексикографската наредба.

4. Сканирайте масива отляво надясно. Всеки път, когато текущото число е различно от предходното, от него започва нова подредица, а всеки път, когато текущото число съвпада с предходното, то е продължение на текущата подредица. Следователно, ако пазите в две променливи **`start`** и **`len`** съответно индекса на началото на текущата подредица от еднакви елементи (в началото той е 0) и дължината на текущата подредица (в началото той е 1), можете да намерите всички подредици от еднакви елементи и техните дължини. От тях лесно може да се избере най-дългата и да се запомня в две допълнителни променливи – **`bestStart`** и **`bestLen`**.

5. Тази задача е много подобна на предходната, но при нея даден елемент се счита за продължение на текущата редица тогава и само тогава, когато е по-голям от предхождащия го елемент.

6. Задачата може да се реши с два вложени цикъла и допълнителен масив **`len[0..n-1]`**. Нека в стойността **`len[i]`** пазим дължината на най-дългата нарастваща подредица, която започва някъде в масива (не е важно къде) и завършва с елемента **`arr[i]`**. Тогава **`len[0]=1`**, а **`len[x]`** е максималната сума **`max(1 + len[prev])`**, където **`prev < x`** и

`arr[prev] < arr[x]`. Следвайки дефиницията `len[0..n-1]` може да се пресметне с два вложени цикъла по следния начин: първият цикъл обхожда масива последователно отляво надясно с водеща променлива `x`. Вторият цикъл (който е вложен в първия) обхожда масива от началото до позиция `x-1` и търси елемент `prev` с максимална стойност на `len[prev]`, за който `arr[prev] < arr[x]`. След приключване на търсенето `len[x]` се инициализира с 1 + най-голямата намерена стойност на `len[prev]` или с 1, ако такава не е намерена.

Описаният алгоритъм намира дължините на всички максимални нарастващи подредици, завършващи във всеки негов елемент. Най-голямата от тези стойности е дължината на най-дългата нарастваща подредица. Ако трябва да намерим самите елементи съставящи тази максимална нарастваща подредица, можем да започнем от елемента, в който тя завършва (нека той е на индекс `x`), да го отпечатаме и да търсим предходния елемент (`prev`). За него е в сила, че `prev < x` и `len[x] = 1+len[prev]`. Така намирайки и отпечатвайки предходния елемент докато има такъв, можем да намерим елементите съставящи най-дългата нарастваща подредица в обратен ред (от последния към първия).

7. Можете да проверите коя от поредица от K числа има най-голяма сума като проверите сумите на всички такива поредици. Първата такава поредица започва от индекс 0 и завършва в индекс K-1 и нека тя има сума S. Тогава втората редица от K елемента започва от индекс 1 и завършва в индекс K, като нейната сума може да се получи като от S се извади нулевия елемент и се добави K-ти елемент. По същия начин може да се продължи до достигане на края на редицата.

8. Потърсете в Интернет информация за алгоритъма "`Selection sort`" и неговите реализации. Накратко идеята е да се намери най-малкият елемент, после да се сложи на първа позиция, след това да се намери втория най-малък и да се сложи на втора позиция и т.н., докато целият масив се подреди в нарастващ ред.

9. Тази задача има два начина, по които може да се реши. Един от тях е с пълно изчерпване, т.е. с два цикъла проверяваме всяка възможна сума. Втория е масива да се обходи само с 1 цикъл като на всяко завъртане на цикъла проверяваме дали текущата сума е по-голяма от вече намерената максимална сума. Задачата може да се реши и с техниката "Динамично оптимиране". Потърсете повече за нея в Интернет.

10. Тази задача може да се решите по много начини. Един от тях е следният: взимате първото число и проверявате колко пъти се повтаря в масива, като пазите този брой в променлива. След всяко прочитане на еднакво число го заменяте с `int.MinValue`. След това взимате следващото и отново правите същото действие. Неговия брой срещания сравнявате с числото, което сте запазили в променливата и ако то е по-голямо, го присвоявате на променливата. Както се

досещате, ако намерите число равно на **`int.MinValue`** преминавате към следващото.

Друг начин да решим задачата е да сортираме числата в нарастващ ред и тогава еднаквите числа ще бъдат разположени като съседни. Така задачата се свежда до намиране на най-дългата редица от съседни числа.

11. Задачата може да се реши с два вложени цикъла. Първият задава началната позиция за втория – от първия до последния елемент. Вторият цикъл започва от позицията, зададена от първия цикъл и сумира последователно числата надясно едно по едно, докато сумата не надвиши S. Ако сумата е равна на S, се запомня числото от първия цикъл (то е началото на поредицата) и числото от втория цикъл (то е краят на поредицата).

    Ако всички числа са положителни, съществува и много по-бърз алгоритъм. Сумирате числата отляво надясно като започвате от нулевото. В момента, в който текущата сума надвиши S, премахвате най-лявото число от редицата и го изваждате от текущата сума. Ако тя пак е по-голяма от търсената, премахвате и следващото число отляво и т.н. докато текущата сума не стане по-малка от S. След това продължавате с поредното число отдясно. Ако намерите търсената сума, я отпечатвате заедно с редицата, която я образува. Така само с едно сканиране на елементите на масива и добавяне на числа от дясната страна към текущата редица и премахване на числа от лявата ѝ страна (при нужда), решавате задачата.

12. Помислете за подходящи начини за итерация върху масивите с два вложени цикъла.

    За d) може да приложите следната стратегия: започвате от позиция (0,0) и се движите надолу N пъти. След това се движите надясно N-1 пъти, след това нагоре N-1 пъти, след това наляво N-2 пъти, след това надолу N-2 пъти и т.н. При всяко преместване слагате в клетката, която напускате поредното число 1, 2, 3, ..., N.

13. Модифицирайте примера за максимална площадка с размер 2 x 2.

14. Задача може да се реши, като се провери за всеки елемент дали като тръгнем по диагонал, надолу или надясно, ще получим поредица. Ако получим поредица проверяваме дали тази поредица е по дълга от предходната най-дълга.

15. Задачата може да решите с масив и два вложени **`for`** цикъла (по буквите на думата и по масива за всяка буква). Задачата има и хитро решение без масив: индексът на дадена главна буква **`ch`** от латинската азбука може да се сметне чрез израза: **`(int) ch – (int) 'A'`**.

16. Потърсете в Интернет информация за алгоритъма "**`binary search`**". Какво трябва да е изпълнено, за да използваме този алгоритъм?

17. Потърсете в Интернет информация за алгоритъма "`merge sort`" и негови реализации.
18. Потърсете в Интернет информация за алгоритъма "`quick sort`" и негови реализации.
19. Потърсете в Интернет информация за "`Sieve of Erathostenes`" (Решетото на Ератостен, учено в часовете по математика).
20. Образувайте всички възможни суми по следния алгоритъм: взимате първото число и го маркирате като "възможна сума". След това взимате следващото подред число и за всяка вече получена "възможна сума" маркирате като възможна сумата на всяка от тях с поредното число. В момента, в който получите числото `S`, спирате с образуването на сумите. Можете да си пазите "възможните суми" или в булев масив където всеки индекс е някоя от сумите, или с по-сложна структура от данни (като `Set` например).
21. Подобна на задача 20 с тази разлика, че ако сумата е равна на `S`, но броя елементи е различен от `K`, продължаваме да търсим. Помислете как да пазите броя числа, с които сте получили определена сума.
22. Задачата може да се реши като се направи допълнителен масив със дължина броя на елементите. В този масив ще пазим текущата най-дълга редица с край елемента на този индекс.
23. Задачата може да се реши с рекурсия, прочетете глава "Рекурсия". Напишете подходяща рекурсия и всеки път променяме позицията на всеки елемент.
24. Подобна на 23 задача.
25. Подобна на 24 задача с тази разлика, че всички елементи в получената комбинация трябва да са във възходящ ред.
26. Помислете за подходящи начини за итерация върху масивите с два вложени цикъла. Разпишете обхождането на лист и помислете как да го реализирате с цикли и изчисления на индексите.
27. Тази задача е доста по-трудна от останалите. Може да използвате алгоритми за обхождане на граф, известни с названията "DFS" (Depth-first-search) или "BFS" (Breadth-first-search). Потърсете информация и примери за тях в Интернет или по-нататък в книгата.

Telerik Academy SCHOOL

Стани част от Академията на Телерик за ученици!

“Основната цел на обучението е да израстнеш, а нашите умове, за разлика от телата ни, могат да растат през целия ни живот!“ Мортимър Адлър

АКАДЕМИЯТА НА ТЕЛЕРИК за ученици предоставя съвременно, специализирано обучение по софтуерно инженерство за ученици от средните училища в България. Нашата основна цел е да помогнем на младите момчета и момичета да се подготвят за Националната олимпиада по информационни технологии (НОИТ) и да придобият солидни теоретични и практически умения за разработка на софтуер и софтуерни технологии.

В Училищната академия на Телерик по софтуерно инженерство:

- **Стимулираме креативността и мисленето на учениците**
- **Всеки ученик получава индивидуално внимание и съдействие**
- **Полагаме основите за писане на качествен програмен код**
- **Изучаваме съвременните софтуерни технологии**
- **Подготвяме учениците за Националната олимпиада по информационни технологии**
- **Предлагаме голямо разнообразие от практически и теоретични обучения**

Програмата на Академията включва:

C#, .NET технологии, бази данни, SQL, Silverlight, 3D графики и моделиране, разработка на игри, програмиране на микроконтролери, WPF, ASP.NET, HTML5, разработка на мобилни приложения, основи на софтуерното инженерство, работа в екип

schoolacademy.telerik.com

groups.google.com/group/it-olymp *facebook.com/TelerikSchoolAcademy*

telerik
deliver more than expected

Глава 8. Бройни системи

В тази тема...

В настоящата тема ще разгледаме работата с различни бройни системи и представянето на числата в тях. Повече внимание ще отделим на представянето на числата в десетична, двоична и шестнадесетична бройна система, тъй като те се използват масово в компютърната техника и в програмирането. Ще обясним и начините за кодиране на числовите данни в компютъра – цели числа без и със знак и различни видове реални числа.

История в няколко реда

Използването на различни бройни системи е започнало още в дълбока древност. Това твърдение се доказва от обстоятелството, че още в Египет са използвани слънчевите часовници, а техните принципи за измерване на времето ползват бройни системи. По-голямата част от историците смятат древноегипетската цивилизация за първата цивилизация, която е разделила деня на по-малки части. Те постигат това, посредством употребата на първите в света слънчеви часовници, които не са нищо друго освен обикновени пръти, забити в земята и ориентирани по дължината и посоката на сянката.

По-късно е изобретен по-съвършен слънчев часовник, който прилича на буквата Т и е градуиран по начин, по който да разделя времето между изгрев и залез слънце на 12 части. Това доказва използването на дванадесетична бройна система в Египет, важността на числото 12 обикновено се свързва и с обстоятелството, че лунните цикли за една година са 12, или с броя на фалангите на пръстите на едната ръка (по три на всеки от четирите пръста, като не се смята палеца).

В днешно време десетичната бройна система е най-разпространената бройна система. Може би това се дължи на улесненията, които тя предоставя на човека, когато той брои с помощта на своите пръсти.

Древните цивилизации са разделили денонощието на по-малки части, като за целта са използвали различни бройни системи, дванадесетични и шестдесетични съответно с основи – 12 и 60. Гръцки астрономи като Хипарх са използвали астрономични подходи, които преди това са били използвани и от вавилонците в Месопотамия. Вавилонците извършвали астрономичните изчисления в шестдесетична система, която били наследили от шумерите, а те от своя страна са я развили около 2000 г. пр. н. е. Не е известно от какви съображения е избрано точно числото 60 за основа на бройната система, но е важно да се знае че, тази система е много подходяща за представяне на дроби, тъй като числото 60 е най-малкото число, което се дели без остатък съответно на 1, 2, 3, 4, 5, 6, 10, 12, 15, 20 и 30.

Някои приложения на шестдесетичната бройна система

Днес шестдесетичната система все още се използва за измерване на ъгли, географски координати и време. Те все още намират приложение при часовниковия циферблат и сферата на глобуса. Шестдесетичната бройна система е използвана и от Ератостен за разделянето на окръжността на 60 части с цел създаване на една ранна система от географски ширини, съставена от хоризонтални линии, минаващи през известни в миналото места от земята. Един век след Ератостен Хипарх нормирал тези линии, като за целта ги направил успоредни и съобразени с геометрията на

Земята. Той въвежда система от линии на географската дължина, в които включват 360 градуса и съответно минават от север до юг и от полюс до полюс. В книгата "Алмагест" (150 г. от н. е.) Клавдий Птолемей доразвива разработките на Хипарх чрез допълнително разделяне на 360-те градуса на географската ширина и дължина на други по-малки части. Той разделил всеки един от градусите на 60 равни части, като всяка една от тези части в последствие била разделена на нови 60 по-малки части, които също били равни. Така получените при деленето части, били наречени partes minutae primae, или "първа минута" и съответно partes minutae secundae, или "втора минута". Тези части се ползват и днес и се наричат съответно "минути" и "секунди".

Кратко обобщение

Направихме кратка историческа разходка през хилядолетията, от която научаваме, че бройните системи са били създадени, използвани и развивани още по времето на шумерите. От изложените факти става ясно защо денонощието съдържа (само) 24 часа, часът съдържа 60 минути, а минутата 60 секунди. Това се дължи на факта, че древните египтяни са разделили по такъв начин денонощието, като са въвели употребата на дванадесетична бройна система. Разделянето на часовете и минутите на 60 равни части, е следствие от работата на древногръцките астрономи, които извършват изчисленията в шестдесетична бройна система, която е създадена от шумерите и използвана от вавилонците.

Бройни системи

До момента разгледахме историята на бройните системи. Нека сега разгледаме какво представляват те и каква е тяхната роля в изчислителната техника.

Какво представляват бройните системи?

Бройните системи (**numeral systems**) са начин за представяне (записване) на числата, чрез краен набор от графични знаци наречени цифри. Към тях трябва да се добавят и правила за представяне на числата. Символите, които се използват при представянето на числата в дадена бройна система, могат да се възприемат като нейна **азбука**.

По време на различните етапи от развитието на човечеството, различни бройни системи са придобивали известност. Трябва да се отбележи, че днес най-широко разпространение е получила арабската бройна система. Тя използва цифрите 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9, като своя азбука. (Интересен е фактът, че изписването на арабските цифри в днешно време се различава от представените по-горе десет цифри, но въпреки това, те пак се отнасят за същата бройна система, т.е. десетичната).

Освен азбука, всяка бройна система има и **основа**. Основата е число, равно на броя различни цифри, използвани от системата за записване на числата в нея. Например арабската бройна система е десетична, защото има 10 цифри. За основа може да се избере произволно число, чиято абсолютна стойност трябва да бъде различна от 0 и 1. Тя може да бъде и реално или комплексно число със знак.

В практическо отношение, възниква въпросът: коя е най-добрата бройна система, която трябва да използваме? За да си отговорим на този въпрос, трябва да решим, как ще се представи по оптимален начин едно число като записване (т.е. брой на цифрите в числото) и брой на цифрите, които използва съответната бройна система, т.е. нейната основа. По математически път, може да се докаже, че най-доброто съотношение между дължината на записа и броя на използваните цифри, се постига при основа на бройната система Неперовото число (*e* = 2,718281828), което е основата на естествените логаритми. Да се работи в система с тази основа, е изключително неудобно, защото това число не може да се представи като отношение на две цели числа. Това ни дава основание да заключим, че оптималната основа на бройната система е 2 или 3. Въпреки, че 3 е по-близо до Неперовото число, то е неподходящо за техническа реализация. Поради тази причина, двоичната бройна система, е единствената подходяща за практическа употреба и тя се използва в съвременните електронноизчислителни машини.

Позиционни бройни системи

Бройните системи се наричат **позиционни** (**positional**), тогава, когато мястото (позицията) на цифрите има значение за стойността на числото. Това означава, че стойността на цифрата в числото не е строго определена и зависи от това на коя позиция се намира съответната цифра в дадено число. Например в числото 351 цифрата 1 има стойност 1, докато при числото 1024 тя има стойност 1000. Трябва да се отбележи, че основите на бройните системи се прилагат само при позиционните бройни системи. В позиционна бройна система числото $A_{(p)} = (a_{(n)}a_{(n-1)}...a_{(0)},a_{(-1)}a_{(-2)}...a_{(-k)})$ може де се представи във вида:

$$A_{(p)} = \sum_{m=n}^{-k} a_m T_m$$

В тази сума T_m има значение на тегловен коефициент за m-тия разряд на числото. В повечето случаи обикновено $T_m = P^m$, което означава, че:

$$A_{(p)} = \sum_{m=n}^{-k} a_m P^m$$

Образувано по горната сума, числото $A_{(p)}$ е съставено съответно от цялата си част $(a_{(n)}a_{(n-1)}...a_{(0)})$ и от дробната си част $(a_{(-1)}a_{(-2)}...a_{(-k)})$, където всяко a принадлежи на множеството от цели числа $M=\{0, 1, 2, ..., p-1\}$. Лесно се вижда, че при позиционните бройни системи стойността на всеки разряд е по-голяма от стойността на предходния разряд (съседния разряд отдясно, който е по-младши) с толкова пъти, колкото е основата на бройната система. Това обстоятелство, налага при събиране да прибавяме единица към левия (по-старшия) разряд, ако трябва да представим цифра в текущия разряд, която е по-голяма от основата. Системите с основи 2, 8, 10 и 16 са получили по-широко разпространение в изчислителната техника, и в следващата таблица е показано съответното представяне на числата от 0 до 15 в тях:

Двоична	**Осмична**	**Десетична**	**Шестнадесетична**
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

Непозиционни бройни системи

Освен позиционни, съществуват и непозиционни бройни системи, при които стойността на всяка цифра е постоянна и не зависи по никакъв начин от нейното място в числото. Като примери за такива бройни системи могат да се посочат съответно римската, гръцката, милетската и др. Като основен недостатък, на непозиционните бройни системи трябва да се посочи това, че чрез тях големите числа се представят неефективно. Заради този си недостатък те са получили по-ограничена употреба. Често това би могло да бъде източник на грешка при определяне на стойността на числата. Съвсем накратко ще разгледаме римската и гръцката бройни системи.

Римска бройна система

Римската бройна система използва следните символи за представяне на числата:

Римска цифра	Десетична равностойност
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Както вече споменахме, в тази бройна система позицията на цифрата не е от значение за стойността на числото, но за нейното определяне се прилагат следните правила:

1. Ако две последователно записани римски цифри, са записани така, че стойността на първата е по-голяма или равна на стойността на втората, то техните стойности се събират. Пример:

   Числото III=3, а числото MMD=2500.

2. Ако две последователно записани римски цифри, са в намаляващ ред на стойностите им, то техните стойности се изваждат. Пример:

   Числото IX=9, числото XML=1040, а числото MXXIV=1024.

Гръцка бройна система

Гръцката бройна система, е десетична система, при която се извършва групиране по петици. Тя използва следните цифри:

Гръцка цифра	Десетична равностойност
I	1
Г	5
Δ	10
Н	100
Х	1 000
М	10 000

Както се вижда в таблицата, единицата се означава с чертичка, петицата с буквата Г, и степените на 10 с началните букви на съответната гръцка дума.

Следват няколко примера на числа от тази система:

- ГΔ = 50 = 5 x 10
- ГН = 500 = 5 x 100
- ГХ = 5000 = 5 x 1 000
- ГМ = 50 000 = 5 x 10 000

Двоичната бройна система – основа на електронноизчислителната техника

Двоичната бройна система (**binary numeral system**), е системата, която се използва за представяне и обработка на числата в съвременните електронноизчислителни машини. Главната причина, поради която тя се е наложила толкова широко, се обяснява с обстоятелството, че устройства с две устойчиви състояния се реализират просто, а разходите за производство на двоични аритметични устройства са много ниски.

Двоичните цифри 0 и 1 лесно се представят в изчислителната техника като "има ток" и "няма ток" или като "+5V" и "-5V".

Наред със своите предимства, двоичната система за представяне на числата в компютъра си има и недостатъци. Един от големите практически недостатъци, е че числата, представени с помощта на тази система са много дълги, т. е. имат голям брой разреди (битове). Това я прави неудобна за непосредствена употреба от човека. За избягване на това неудобство, в практиката се ползват бройни системи с по-големи основи.

Десетични числа

Числата представени в **десетична бройна система** (**decimal numeral system**), се задават в първичен вид, т.е. вид удобен за възприемане от човека. Тази бройна система има за основа числото 10. Числата записани в нея са подредени по степените на числото 10. Младшият разряд (първият отдясно на ляво) на десетичните числа се използва за представяне на единиците (10^0=1), следващият за десетиците (10^1=10), следващият за стотиците (10^2=100) и т.н. Казано с други думи, всеки следващ разряд е десет пъти по-голям от предшестващия го разряд. Сумата от отделните разряди определя стойността на числото. За пример ще вземем числото 95031, което в десетична бройна система се представя като:

$$95031 = (\mathbf{9}\times10^4) + (\mathbf{5}\times10^3) + (\mathbf{0}\times10^2) + (\mathbf{3}\times10^1) + (\mathbf{1}\times10^0)$$

Представено в този вид, числото 95031 е записано по естествен за човека начин, защото принципите на десетичната система са възприети като фундаментални за хората.

Разгледаните подходи важат и за останалите бройни системи. Те имат същата логическа постановка, но тя е приложена за бройна система с друга основа. Последното твърдение, се отнася включително и за двоичната и шестнайсетината бройни системи, които ще разгледаме в детайли след малко.

Двоични числа

Числата представени в двоична бройна система, се задават във вторичен вид, т.е. вид удобен за възприемане от изчислителната машина. Този вид е малко по-трудно разбираем за човека. За представянето на двоичните числа, се използва двоичната бройна система, която има за основа числото 2. Числата записани в нея са подредени по степените на двойката. За тяхното представяне, се използват само цифрите 0 и 1.

Прието е, когато едно число се записва в бройна система, различна от десетичната, във вид на индекс в долната му част да се отразява, коя бройна система е използвана за представянето му. Например със записа $\mathbf{1110}_{\mathbf{(2)}}$ означаваме число в двоична бройна система. Ако не бъде указана изрично, бройната система се приема, че е десетична. Числото се произнася, като се прочетат последователно неговите цифри, започвайки от ляво на дясно (т.е. прочитаме го от старшия към младшия разряд "бит").

Както и при десетичните числа, гледано от дясно наляво, всяко двоично число изразява степените на числото 2 в съответната последователност. На младшата позиция в двоично число съответства нулевата степен (2^0=1), на втората позиция съответства първа степен (2^1=2), на третата позиция съответства втора степен (2^2=4) и т.н. Ако числото е 8-битово, степените достигат до седма (2^7=128). Ако числото е 16-битово, степените достигат до петнадесета (2^{15}=32768). Чрез 8 двоични цифри (0 или 1) могат да се представят общо 256 числа, защото 2^8=256. Чрез 16 двоични цифри могат да се представят общо 65536 числа, защото 2^{16}=65536.

Нека дадем един пример за числа в двоична бройна система. Да вземем десетичното число **148**. То е съставено от три цифри: **1**, **4** и **8**, и съответства на следното двоично число:

$$\mathbf{10010100}_{\mathbf{(2)}}$$

$$\mathbf{148} = (\mathbf{1}\times 2^7) + (\mathbf{1}\times 2^4) + (\mathbf{1}\times 2^2)$$

Пълното представяне на това число е изобразено в следващата таблица:

Число	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**
Степен	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Стойност	1×2^7 = 128	0×2^6 = 0	0×2^5 = 0	1×2^4 = 16	0×2^3 = 0	1×2^2 = 4	0×2^1 = 0	0×2^0 = 0

Последователността от осем на брой нули и единици представлява един **байт**, т.е. това е едно обикновено осем-разрядно двоично число. Чрез един байт могат да се запишат всички числа от 0 до 255 включително. Много често това е не достатъчно и затова се използват по няколко последователни байта за представянето на едно число. Два байта образуват т. н. "машинна дума" (**word**), която отговаря на 16 бита (при 16-разредните изчислителни машини). Освен нея, в изчислителните машини се използва и т.н. "двойна дума" (**double word**) или (**dword**), съответстваща на 32 бита.

Ако едно двоично число завършва на 0, то е четно, а ако завършва на 1, то е нечетно.

Преминаване от двоична в десетична бройна система

При преминаване от двоична в десетична бройна система, се извършва преобразуване на двоичното число в десетично. Всяко число може да се преобразува от една бройна система в друга, като за целта се извършат последователност от действия, които са възможни и в двете бройни системи. Както вече споменахме, числата записани в двоична бройна система се състоят от двоични цифри, които са подредени по степените на двойката. Нека да вземем за пример числото $\mathbf{11001}_{\mathbf{(2)}}$. Преобразуването му в десетично се извършва чрез пресмятането на следната сума:

$$\mathbf{11001}_{(2)} = \mathbf{1}\times 2^4 + \mathbf{1}\times 2^3 + \mathbf{0}\times 2^2 + \mathbf{0}\times 2^1 + \mathbf{1}\times 2^0 =$$

$$= \mathbf{16}_{(10)} + \mathbf{8}_{(10)} + \mathbf{1}_{(10)} = \mathbf{25}_{(10)}$$

От това следва, че $\mathbf{11001}_{(2)} = \mathbf{25}_{(10)}$

С други думи, всяка една двоична цифра се умножава по 2 на степен, позицията, на която се намира (в двоичното число). Накрая се извършва събиране на числата, получени за всяка от двоичните цифри, за да се получи десетичната равностойност на двоичното число.

Съществува и още един начин за преобразуване, който е известен като схема на Хорнер. При тази схема се извършва умножение на най-лявата цифра по две и събиране със съседната ѝ вдясно. Този резултат се умножава по две и се прибавя следващата съседна цифра от числото (цифрата вдясно). Това продължава до изчерпване на всички цифри в числото, като последната цифра от числото се добавя без умножаване. Ето един пример:

$$\mathbf{1001}_{(2)} = ((1 \times 2 + 0) \times 2 + 0) \times 2 + 1 = 2 \times 2 \times 2 + 1 = \mathbf{9}$$

Преминаване от десетична към двоична бройна система

При преминаване от десетична в двоична бройна система, се извършва преобразуване на десетичното число в двоично. За целите на преобразуването се извършва делене на две с остатък. Така се получават частно и остатък, който се отделя.

Отново ще вземем за пример числото 148. То се дели целочислено на основата, към която ще преобразуваме (в примера тя е 2). След това, от остатъците получени при деленето (те са само нули и единици), се записва преобразуваното число. Деленето продължава, докато получим частно нула. Ето пример:

148:2=74 имаме остатък 0;

74:2=37 имаме остатък 0;

37:2=18 имаме остатък 1;

18:2=9 имаме остатък 0;

9:2=4 имаме остатък 1;

4:2=2 имаме остатък 0;

2:2=1 имаме остатък 0;

1:2=0 имаме остатък 1;

След като вече семе извършили деленето, записваме стойностите на остатъците в ред, обратен на тяхното получаване, както следва:

10010100

т.е. $148_{(10)} = 10010100_{(2)}$

Действия с двоични числа

При двоичните числа за един двоичен разряд са в сила аритметичните правила за събиране, изваждане и умножение:

0 + 0 = 0	0 - 0 = 0	0 × 0 = 0
1 + 0 = 1	1 - 0 = 1	1 × 0 = 0
0 + 1 = 1	1 - 1 = 0	0 × 1 = 0
1 + 1 = 10	10 - 1 = 1	1 × 1 = 1

С двоичните числа могат да се извършват и логически действия, като логическо умножение (конюнкция), логическо събиране (дизюнкция) и сума по модул две (изключващо или).

Трябва да се отбележи, че при извършване на аритметични действия над многоразредни числа трябва да се отчита връзката между отделните разреди чрез пренос или заем, когато извършваме съответно събиране или изваждане. Да разгледаме някои детайли относно побитовите оператори.

Побитово "и"

Побитов **AND** оператор – може да се използва за проверка на стойност на даден бит в число. Например, ако искаме да проверим дали дадено число е четно (проверяваме дали най-младшият бит е **1**):

1011101**1** **AND** 0000000**1** = 00000001

Резултатът е 1 и това означава, че числото е нечетно (ако резултатът беше 0, значи е четно).

В C# побитовото "и" се означава с **&** и се използва така:

```
int result = integer1 & integer2;
```

Побитово "или"

Побитов **OR** оператор – може да се ползва, ако например искаме да "вдигнем" даден бит в 1:

10111**0**11 **OR** 00000**1**00 = 10111111

Означението на побитовото "или" в C# е **|** и се използва така:

```
int result = integer1 | integer2;
```

Побитово "изключващо и"

Побитов **XOR** оператор – всяка двоична цифра се обработва поотделно, като когато имаме 0 във втория операнд, стойността на същия бит от първия се копира в резултата. Където имаме 1 във втория операнд, там обръщаме стойността от съответната позиция на първия и записваме в резултата:

10111011 **XOR** 01010101 = 11101110

В C# означението на оператора "изключващо или" е **^**:

```
int result = integer1 ^ integer2;
```

Побитово отрицание

Побитов **NOT** оператор – това е унарен (unary) оператор, което означава, че се прилага върху един единствен операнд. Това, което прави е да обърне всеки бит от дадено двоично число в обратната стойност:

NOT 10111011 = 01000100

В C# побитовото отрицание се отбелязва с **~**:

```
int result = ~integer1;
```

Шестнайсетични числа

При **шестнайсетичните числа** (**hexadecimal numbers**) имаме за основа на бройната система числото 16, което налага да бъдат използвани 16 знака (цифри) за представянето на всички възможни стойности от 0 до 15 включително. Както вече беше показано в една от таблиците в предходните точки, за представянето на шестнайсетичните числа се използват числата от 0 до 9 и латинските букви от A до F. Всяка от тях има съответната стойност:

A=10, B=11, C=12, D=13, E=14, F=15

Като примери за шестнайсетични числа могат да бъдат посочени съответно, D2, 1F2F1, D1E и др.

Преминаването към десетична система става като се умножи по 16^0 стойността на най-дясната цифра, по 16^1 следващата вляво, по 16^2 следващата вляво и т.н. и накрая се съберат. Например:

$D1E_{(16)} = E*16^0 + 1*16^1 + D*16^2 = 14*1 + 1*16 + 13*256 = 3358_{(10)}$.

Преминаването от десетична към шестнайсетична бройна система става като се дели десетичното число на 16 и се вземат остатъците в обратен ред. Например:

3358 / 16 = 209 + остатък 14 (E)

209 / 16 = 13 + остатък 1 (1)

13 / 16 = 0 + остатък 13 (D)

Взимаме остатъците в обратен ред и получаваме числото $D1E_{(16)}$.

Бързо преминаване от двоични към шестнайсетични числа

Бързото преобразуване, от двоични в шестнайсетични числа се извършва бързо и лесно, чрез разделяне на двоичното число на групи от по четири бита (разделяне на полубайтове). Ако броят на цифрите в числото не е кратен на четири, то се добавят водещи нули в старшите разряди. След разделянето и евентуалното добавяне на нули, се заместват всички получени групи със съответстващите им цифри. Ето един пример:

Нека да ни е дадено следното число: $1110011110_{(2)}$.

1. Разделяме го на полубайтове и добавяме водещи нули

   Пример: 0011 1001 1110.

2. Заместваме всеки полубайт със съответната шестнайсетична цифра и така получаваме $39E_{(16)}$.

Следователно $1110011110_{(2)} = 39E_{(16)}$.

Бройни системи – обобщение

Като обобщение ще формулираме отново, в кратък, но ясен вид алгоритмите за преминаване от една позиционна бройна система в друга:

- Преминаването от десетична в k-ична бройна система се извършва като последователно се дели десетичното число на основата на новата система k и получените остатъци (съответстващата им k-ична цифра) се натрупват в обратен ред.
- Преминаването от k-ична бройна система към десетична се извършва като се умножи последната цифра на k-ичното число по k^0, предпоследната – по k^1, следващата – по k^2 и т.н. и получените произведения се сумират.
- Преминаването от k-ична бройна система към p-ична се извършва чрез междинно преминаване към десетична бройна система (с изключение на случая шестнайсетична / двоична бройна система).

Представяне на числата

За съхраняване на данните в оперативната памет на електронноизчислителните машини се използва двоичен код. В зависимост от това какви данни съхраняваме (символи, цели или реални числа с цяла и дробна част) информацията се представя по различен начин. Този начин се определя от типа на данните.

Дори и програмистът на език от високо ниво трябва да знае, какъв вид имат данните разположени в оперативната памет на машината. Това се отнася, и за случаите, когато данните се намират на външен носител, защото при обработката им те се разполагат в оперативната памет.

В настоящата секция са разгледани начините за представяне и обработка на различни типове данни. Най-общо те се основават на понятията бит, байт и машинна дума.

Бит е една двоична единица от информация, със стойност 0 или 1.

Информацията в паметта се групира в последователности от 8 бита, които образуват един **байт**.

За да бъдат обработени от аритметичното устройство, данните се представят в паметта от определен брой байтове (2, 4 или 8), които образуват машинната дума. Това са концепции, които всеки програмист трябва задължително да знае и разбира.

Представяне на цели числа в паметта

Едно от нещата, на които до сега не обърнахме внимание, е знакът на числата. Представянето на целите числа в паметта на компютъра може да се извърши по два начина: със знак или без знак. Когато числата се представят със знак се въвежда знаков разред. Той е най-старшият разред и има стойност 1 за отрицателните числа и 0 за положителните. Останалите разреди са информационни и отразяват (съдържат) стойността на числото. В случая на числа без знак всички битове се използват за записване на стойността на числото.

Цели числа без знак

За **целите числа без знак** (**unsigned integers**) се заделят по 1, 2, 4 или 8 байта от паметта. В зависимост, от броя на байтовете използвани при представянето на едно число, се образуват обхвати на представяне с различна големина. Посредством *n* на брой бита могат да се представят цели числа без знак в обхвата [0, 2^n-1]. Следващата таблица, показва обхвата от стойности на целите числа без знак:

Брой байтове за представяне на числото в паметта	**Обхват**	
	Запис чрез порядък	**Обикновен запис**
1	0 ÷ 2^8-1	0 ÷ 255
2	0 ÷ 2^{16}-1	0 ÷ 65 535
4	0 ÷ 2^{32}-1	0 ÷ 4 294 967 295
8	0 ÷ 2^{64}-1	0 ÷ 18 446 744 073 709 551 615

Ще покажем пример при еднобайтово и двубайтово представяне на числото 158, което се записва в двоичен вид като $10011110_{(2)}$:

1. Представяне с 1 байт:

1	0	0	1	1	1	1	0

2. Представяне с 2 байта:

0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	0

Представяне на отрицателни числа

За отрицателните числа се заделят по 1, 2, 4 или 8 байта от паметта на компютъра, като най-старшият разред (най-левия бит) има значение на знаков и носи информация за знака на числото. Както вече споменахме, когато знаковият бит има стойност 1 числото е отрицателно, а в противен случай е положително.

Следващата таблица, показва обхвата от стойности на целите числа със знак в компютърната техника според броя байтове, използвани за записването им:

Брой байтове за представяне на числото в паметта	Обхват	
	Запис чрез порядък	Обикновен запис
1	$-2^{7} \div 2^{7}-1$	-128 ÷ 127
2	$-2^{15} \div 2^{15}-1$	-32 768 ÷ 32 767
4	$-2^{31} \div 2^{31}-1$	-2 147 483 648 ÷ 2 147 483 647
8	$-2^{63} \div 2^{63}-1$	-9 223 372 036 854 775 808 ÷ 9 223 372 036 854 775 807

За кодирането на отрицателните числа, се използват прав, обратен и допълнителен код. И при трите представяния целите числа със знак са в границите: $[-2^{n-1}, 2^{n-1}-1]$. Положителните числа винаги се представят по един и същи начин и за тях правият, обратният и допълнителният код съвпадат.

Правият код (**signed magnitude**) е най-простото представяне на числото. Старшият бит е знаков, а в оставащите битове е записана абсолютната стойност на числото. Ето няколко примера:

Числото 3 в прав код се представя в осембитово число като 00000011.

Числото -3 в прав код се представя в осембитово число като 10000011.

Обратният код (**one's complement**) се образува от правия код на числото, чрез инвертиране (заместване на всички нули с единици и единици с нули). Този код не е никак удобен за извършването на аритметичните действия събиране и изваждане, защото се изпълнява по различен начин, когато се налага изваждане на числа. Освен това се налага знаковите битове да се обработват отделно от информационните. Този недостатък се избягва с употребата на допълнителен код, при който вместо изваждане се извършва събиране с отрицателно число. Последното е представено чрез неговото допълнение, т.е. разликата между 2^{n} и самото число. Пример:

Числото -127 в прав код се представя като 1 1111111, а в обратен код като 1 0000000.

Числото 3 в прав код се представя като 0 0000011, а в обратен код има вида 0 1111100.

Допълнителният код (**two's complement**) е число в обратен код, към което е прибавена (чрез събиране) единица. Пример:

Числото -127 представено в допълнителен код има вида 1 0000001.

При **двоично-десетичния код**, известен е още като **BCD** код (**Binary Coded Decimal**) в един байт се записват по две десетични цифри. Това се постига, като чрез всеки полубайт се кодира една десетична цифра. Числа представени чрез този код могат да се пакетират, т.е. да се представят в пакетиран формат. Ако представим една десетична цифра в един байт се получава непакетиран формат.

Съвременните микропроцесори използват един или няколко от разгледаните кодове за представяне на отрицателните числа, като най-разпространеният начин е представянето в допълнителен код.

Целочислени типове в C#

В C# има осем целочислени типа данни със знак и без знак. В зависимост от броя байтове, които се заделят в паметта за тези типове, се определя и съответният диапазон от стойности, които те могат да заемат. Следват описания на типовете:

Тип	Размер	Обхват	Тип в .NET framework
`sbyte`	8 бита	-128 ÷ 127	**`System.SByte`**
`byte`	8 бита	0 ÷ 255	**`System.Byte`**
`short`	16 бита	-32,768 ÷ 32,767	**`System.Int16`**
`ushort`	16 бита	0 ÷ 65,535	**`System.UInt16`**
`int`	32 бита	-2,147,483,648 ÷ 2,147,483,647	**`System.Int32`**
`uint`	32 бита	0 ÷ 4,294,967,295	**`System.UInt32`**
`long`	64 бита	–9,223,372,036,854,775,808 ÷ 9,223,372,036,854,775,807	**`System.Int64`**
`ulong`	64 бита	0 ÷ 18,446,744,073,709,551,615	**`System.UInt64`**

Ще разгледаме накратко най-използваните типове. Най-широко използваният целочислен тип е **`int`**. Той се представя като 32-битово число в допълнителен код и приема стойности в интервала [-2^{31}, 2^{31}-1]. Променливите от този тип най-често се използват за управление на цикли, индексиране на масиви и други целочислени изчисления. В следващата таблица е даден пример за декларация на променлива от тип **`int`**:

```csharp
int integerValue = 25;
int integerHexValue = 0x002A;
int y = Convert.ToInt32("1001", 2); // Converts binary to int
```

Типът **`long`** е най-големият целочислен тип със знак в C#. Той има размерност 64 бита (8 байта). При присвояване на стойности на променливите от тип **`long`** се използват латинските букви "**`l`**" или "**`L`**", които се поставят в края на целочисления литерал. Поставен на това място, този модификатор означава, че литералът има стойност от тип **`long`**. Това се прави, защото по подразбиране целочислените литерали са от тип **`int`**. В следващия пример декларираме и присвояваме 64-битови цели числа на променливи от тип **`long`**:

```
long longValue = 9223372036854775807L;
long newLongValue = 932145699054323689l;
```

Важно условие е да се внимава да не бъде надхвърлен обхватът на представимите числа и за двата типа. Все пак C# предоставя възможността да контролираме какво се случва когато настъпи препълване. Това става посредством **`checked`** и **`unchecked`** блоковете. Първите се използват когато е нужно приложението да хвърли изключение (от тип **`System. OverflowException`**) в случай на надхвърляне на обхвата на променливата. Следният програмен код прави именно това:

```
checked
{
   int a = int.MaxValue;
   a = a + 1;
   Console.WriteLine(a);
}
```

В случай че фрагментът е в **`unchecked`** блок, изключение не се хвърля и изведеният резултат е неверен:

```
-2147483648
```

По подразбиране, в случай че не се използват тези блокове C# компилаторът работи в unchecked режим.

C# включва и типове без знак, които могат да бъдат полезни при нужда от по-голям обхват на променливите в диапазона на положителните числа. По-долу са няколко примера за деклариране на променливи без знак. Обърнете внимание на суфиксите за **`ulong`** (всякакви комбинации от **`U`**, **`L`**, **`u`**, **`l`**).

```
byte count = 50;
ushort pixels = 62872;
uint points = 4139276850; // or 4139276850u, 4139276850U
ulong y = 18446744073709551615; // or UL, ul, Ul, uL, Lu, lU
```

Представянията Big-Endian и Little-Endian

При цели числа, които се записват в повече от един байт, има два варианта за наредба на байтовете в паметта:

- Little-Endian (LE) – байтовете се подреждат от ляво надясно от най-младшия към най-старшия. Това представяне се използва при Intel x86 и Intel x64 микропроцесорните архитектури.
- Big-Endian (BE) – байтовете се подреждат от ляво надясно от най-старшия към най-младшия. Това представяне се използва при PowerPC, SPARC и ARM микропроцесорните архитектури.

Ето един пример: числото A8B6EA72$_{(16)}$ се представя в двете наредби на байтовете по следния начин:

`0x72`	`0xEA`	`0xB6`	`0xA8`

Little-Endian (LE)
for 0xA8B6EA72

`0xA8`	`0xB6`	`0xEA`	`0x72`

Big-Endian (BE)
for 0xA8B6EA72

Има някои класове в C#, които предоставят възможности за дефиниране на това кой стандарт за подредба на байтовете да се използва. Това е важно при операции от като изпращане / приемане на потоци от информация по мрежата и други видове комуникация между устройства, произведени по различни стандарти. Полето **`IsLittleEndian`** на **`BitConverter`** класа например показва в какъв режим класът работи и как се съхраняват данните за текущата компютърна архитектура.

Представяне на реални числа с плаваща запетая

Реалните числа са съставени от цяла и дробна част. В компютърната техника, те се представят като **числа с плаваща запетая** (**floating-point numbers**). Всъщност това представяне идва от възприетия от водещите производители на микропроцесори Standard for Floating-Point Arithmetic (IEEE 754). Повечето хардуерни платформи и езици за програмиране позволяват или изискват изчисленията да се извършват съгласно изискванията на този стандарт. Стандартът определя:

- Аритметични формати: набор от двоични и десетични данни с плаваща запетая, които са съставени от краен брой цифри.
- Формати за обмен: кодировки (битови низове), които могат да бъдат използвани за обмен на данни в една ефективна и компактна форма.
- Алгоритми за закръгляване: методи, които се използват за закръгляване на числата по време на изчисления.
- Операции: аритметика и други операции на аритметичните формати.
- Изключения: представляват сигнали за извънредни случаи като деление на нула, препълване и др.

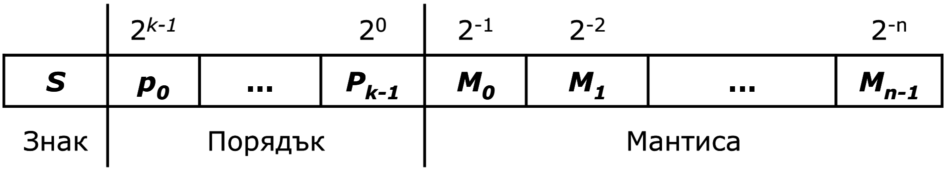
Съгласно IEEE-754 стандарта произволно реално число ***R*** може да бъде представено във вида:

$$R = M * q^p$$

където ***M*** е **мантисата** на числото, а ***p*** е **порядъкът** му (**експонента**), и съответно ***q*** е основа на бройната система, в която е представено числото. Мантисата трябва да бъде положителна или отрицателна правилна дроб, т.е. $|M|<1$, а порядъкът – положително или отрицателно цяло число.

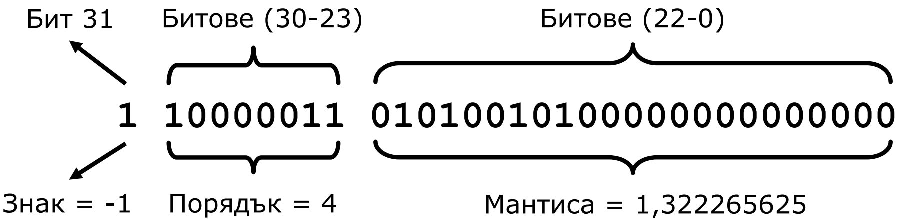
При посочения начин на представяне на числата, всяко число във формат с плаваща запетая, ще има следния обобщен вид $\pm 0,M*q^{\pm p}$.

В частност, когато представяме числата във формат с плаваща запетая чрез двоична бройна система, ще имаме $R = M * 2^p$. При това представяне на реалните числа в паметта на компютъра, след промяна на порядъка се стига и до изместване "плаване" на десетичната запетая в мантисата. Форматът на представянето с плаваща запетая има полулогаритмична форма. Той е изобразен нагледно на следващата фигура:



Представяне на числа с плаваща запетая – пример

Нека дадем един пример за представяне на число с плаваща запетая в паметта. Искаме да запишем числото -21,15625 в 32-битов (single precision) floating-point формат по стандарта IEEE-754. При този формат се използват 23 бита за мантиса, 8 бита за експонента и 1 бит за знак на числото. Представянето на числото е следното:



Знакът на числото е отрицателен, т. е. мантисата има отрицателен знак:

$$S = -1$$

Порядъкът (експонентата) има стойност 4 (записана с изместен порядък):

$$p = (2^0 + 2^1 + 2^7) - 127 = (1+2+128) - 127 = 4$$

За преминаване към истинската стойност изваждаме 127 от стойността на допълнителния код, защото работим с 8 бита ($127 = 2^8-1$).

Мантисата има следната стойност (без да взимаме предвид знака):

$$M = 1 + 2^{-2} + 2^{-4} + 2^{-7} + 2^{-9} =$$
$$= 1 + 0{,}25 + 0{,}0625 + 0{,}0078125 + 0{,}001953125 =$$
$$= 1{,}322265625$$

Забелязахте ли, че добавихме единица, която липсва в двоичния запис на мантисата? Това е така, защото мантисата винаги е нормализирана и започва с единица, която се подразбира.

Стойността на числото се изчислява по формулата $R = M * 2^p$, която в нашия пример добива вида:

$$R = -1{,}3222656 * 2^4 = -1{,}322265625 * 16 = -21{,}1562496 \approx -21{,}15625$$

Нормализация на мантисата

За по-пълното използване на разрядната решетка мантисата трябва да съдържа единица в най-старшия си разред. Всяка мантиса удовлетворяваща това условие са нарича **нормализирана**. При IEEE-754 стандарта единицата в цялата част на мантисата се подразбира, т.е. мантисата е винаги число между 1 и 2.

Ако по време на изчисленията се достигне до резултат, който не удовлетворява това условие, то имаме нарушение на нормализацията. Това изисква, преди да се пристъпи към по-нататъшна обработка на числото то да бъде нормализирано, като за целта се измества запетаята в мантисата и след това се извършва съответна промяна на порядъка.

Типовете float и double в C#

В C# разполагаме с два типа данни за представяне на числата с плаваща запетая. Типът `float` е 32-битово реално число с плаваща запетая, за което е прието да се казва, че има единична точност (single precision floating-point number). Типът `double` е 64-битово реално число с плаваща запетая, за което е прието да се казва, че има двойна точност (double precision floating-point number). Тези реални типове данни и аритметичните операции върху тях съответстват на спецификацията, определена от стандарта IEEE 754-1985. В следната таблица са по-важните характеристики на двата типа:

Тип	Размер	Обхват	Значещи цифри	Тип в .NET framework
`float`	32 бита	$\pm1.5 \times 10^{-45} \div \pm3.4 \times 10^{38}$	7	`System.Single`
`double`	64 бита	$\pm5.0 \times 10^{-324} \div \pm1.7 \times 10^{308}$	15-16	`System.Double`

При тип `float` имаме мантиса, която съхранява 7 значещи цифри, докато при тип `double` тя съхранява 15-16 значещи цифри. Останалите битове се използват за задаването на знаците на мантисата и стойността на порядъка. Типът `double` освен с по-голям брой значещи цифри разполага и с по-голям порядък, т.е. обхват на приеманите стойности. Ето един пример за декларация на променливи от тип `float` и тип `double`:

```
float total = 5.0f;
float result = 5.0f;
double sum = 10.0;
double div = 35.4 / 3.0;
double x = 5d;
```

Суфиксите, поставени след числата от дясната страна на равенството са с цел те да бъдат третирани като числа от съответен тип (`f` за `float`, `d` за `double`). В случая са поставени, тъй като по подразбиране 5.0 ще се интерпретира от тип `double`, а 5 – от тип `int`.

В C# по подразбиране числата с плаваща запетая, въведени като литерал са от тип `double`.

Целочислени и числа с плаваща запетая могат да присъстват в даден израз. В такъв случай целочислените променливи биват конвертирани към такива с плаваща запетая и резултатът се получава по следните правила:

1. Ако някои от типовете с плаваща запетая е `double`, резултатът е от тип `double` (или `bool`).
2. Ако няма `double` тип в израза, резултатът е от тип `float` (или `bool`).

Много от математическите операции могат да дадат резултати, които нямат конкретна числена стойност като например стойността "+/- безкрайност" или стойността `NaN` (което означава "Not a Number"), които не представляват числа. Ето един пример:

```
double d = 0;
Console.WriteLine(d);
Console.WriteLine(1/d);
Console.WriteLine(-1/d);
Console.WriteLine(d/d);
```

Ако го изпълним, ще получим следния резултат:

```
0.0
Infinity
-Infinity
NaN
```

Ако изпълним горния код с тип `int` вместо `double`, ще получим `System.DivideByZeroException`, защото целочисленото деление на 0 е непозволена операция.

Грешки при числата с плаваща запетая

Числата с плаваща запетая (представени съгласно стандарта IEEE 754) са удобни за работа с изчисления от физиката, където се използват много големи числа (с няколко стотици цифри) и много близки до нулата числа (със стотици цифри след десетичната запетая преди първата значеща цифра). При такива числа форматът IEEE 754 е изключително удобен, защото запазва порядъка на числото в експонентата, а мантисата се ползва само за съхранение на значещите цифри. При 64-битови числа с плаваща запетая се постига точност до 15-16 цифри и експонента отместваща десетичната точка над 300 позиции наляво и надясно.

За съжаление не всяко реално число има точно представяне във формат IEEE 754, тъй като не всяко число може да се представи като полином на ограничен брой събираеми, които са отрицателни степени на двойката. Това важи с пълна сила дори за числата, които употребяваме ежедневно при най-простите финансови изчисления. Например числото 0.1 записано като в 32-битова floating-point стойност се представя като 0.099999994. Ако се ползва подходящо закръгляне, числото се възприема като 0.1, но грешката може да се натрупа и да даде сериозни отклонения, особено при финансови изчисления. Например при сумиране на 1000 артикула с единична цена от по 0.1 EUR би трябвало да се получи сума 100 EUR, но ако смятаме с 32-битови floating-point числа, ще получим сумата 99.99905. Ето реален пример на C# в действие, който доказва грешките причинени от неточното представяне на десетичните реални числа в двоична бройна система:

```
float sum = 0f;
for (int i = 0; i < 1000; i++)
{
   sum += 0.1f;
}
Console.WriteLine("Sum = {0}", sum);
// Sum = 99.99905
```

Можете сами да се убедите в грешките при подобни пресмятания като изпълните примера или си поиграете с него и го модифицирате, за да получите още по-фрапантни грешки.

Точност на числата с плаваща запетая

Точността на резултатите от изчисленията при работа с числа с плаваща запетая зависят от следните параметри:

1. Точността на представяне на числата.
2. Точността на използваните числени методи.
3. Стойностите на грешките, получени при закръгляване и др.

Поради това, че числата се представят в паметта с някаква точност, при изчисленията върху тях резултатите могат също да са неточни. Нека като пример да разгледаме следния програмен фрагмент:

```
double sum = 0.0;
for (int i = 1; i <= 10; i++)
{
   sum += 0.1;
}
Console.WriteLine("{0:r}", sum);
Console.WriteLine(sum);
```

По време на неговото изпълнение, в цикъл добавяме стойността 1/10 в променливата `sum`. В извикването на метода **`WriteLine()`** използваме round-trip format спецификаторът **`"{0:r}"`** за да изведем точната (незакръглена) стойност която се съдържа в променливата, а след това отпечатваме същата стойност без да указваме формат. Очаква се, че при изпълнението на тази програма ще получим резултат 1.0, но в действителност, когато закръглянето е изключено програмата извежда стойност, много близка до вярната, но все пак различна:

```
0.99999999999999989
1
```

Както се вижда от примера, по подразбиране при отпечатване на floating-point числа в .NET Framework те се закръглят и така привидно се намаляват грешките от неточното им представяне във формата IEEE 754. Резултатът от горното изчисление, както се вижда, е грешен, но след закръгляне изглежда правилен. Ако обаче съберем 0.1 няколко хиляди пъти, грешката ще се натрупа и закръглянето не може да я компенсира.

Причината за грешния резултат в примера е, че числото 0.1 няма точно представяне в типа **`double`** и се представя със закръгляне. Нека заменим **`double`** с **`float`**:

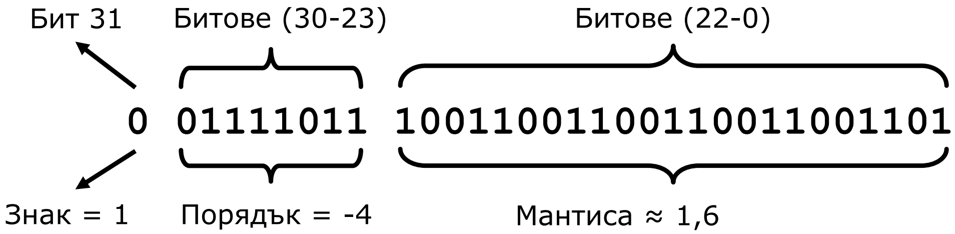
```
float sum = 0.0f;
for (int i = 1; i <= 10; i++)
{
   sum += 0.1f;
}
Console.WriteLine("{0:r}", sum);
```

Ако изпълним горния код ще получим съвсем друга сума:

```
1.00000012
```

Причината за това отново е закръглянето.

Ако направим разследване защо се получават тези резултати, ще се убедим, че числото 0.1 се представя в типа **`float`** по следния начин:



Всичко изглежда коректно с изключение на мантисата, която има стойност, малко по-голяма от 1.6, а не точно 1.6, защото това число не може да се представи като сума от отрицателни степени на 2. Ако трябва да сме съвсем точни, стойността на мантисата е 1 + 1 / 2 + 1 / 16 + 1 / 32 + 1 / 256 + 1 / 512 + 1 / 4096 + 1 / 8192 + 1 / 65536 + 1 / 131072 + 1 / 1048576 + 1 / 2097152 + 1 / 8388608 ≈ 1,60000002384185791015625 ≈ 1.6. Така числото 0.1 в крайна сметка се представя във формат IEE 754 като съвсем малко повече от 1.6 × 2^{-4} и грешката настъпва не при събирането, а още преди това – при записването на 0.1 в типа `float`.

Типовете `double` и `float` имат поле `Epsilon`, което е константа и съдържа най-малката стойност по-голяма от 0, която съответно `System.Single` или `System.Double` инстанцията може да представи. Всяка стойност по-малка от `Epsilon` се счита за равна на 0. Така например, ако сравняваме две числа, които са все пак различни, но тяхната разлика е по-малка от `Epsilon`, то те ще бъдат сметнати за равни.

Типът decimal

Типът `System.Decimal` в .NET Framework използва **десетична аритметика с плаваща запетая** (**decimal floating-point arithmetic**) и 128-битова точност, която е подходяща за големи и прецизни финансови изчисления. Ето и някои характеристики на типа `decimal`:

Тип	Размер	Обхват	Значещи цифри	Тип в .NET framework
`decimal`	128 бита	$\pm1.0 \times 10^{-28} \div$ $\pm7.9 \times 10^{28}$	28-29	`System.Decimal`

За разлика от числата с плаваща запетая, типът `decimal` запазва точност за всички десетични числа, които са му в обхвата. Тайната за тази отлична точност при работа с десетични числа се крие във факта, че вътрешното представяне на мантисата не е в двоична бройна система, а в десетична. Експонентата му също е степен на 10, а не на 2. Така не се налага приближено представяне на десетичните числа – те се записват точно, без преобразуване в двоична бройна система.

Тъй като типовете `float` и `double` и операциите върху тях се реализират хардуерно от аритметичния копроцесор, който е част от всички съвременни компютърни микропроцесори, а `decimal` се реализира софтуерно, типът **`decimal`** е няколко десетки пъти по-бавен от **`double`**, но е незаменим при изпълнението на финансови изчисления.

В случай, че целим да присвоим даден литерал на променлива от тип **`decimal`**, е нужно да използваме суфиксите **`m`** или **`M`**. Например:

```
decimal calc = 20.4m;
decimal result = 5.0M;
```

Нека използваме **`decimal`** вместо **`float`** / **`double`** в примера, който разгледахме преди малко:

```
decimal sum = 0.0m;
for (int i = 1; i <= 10000000; i++)
{
   sum += 0.0000001m;
}
Console.WriteLine(sum);
```

Резултатът този път е точно такъв, какъвто се очаква да бъде:

```
1.0000000
```

Въпреки че **`decimal`** типът има по-голяма точност, от типовете с плаваща запетая, той предоставя по-малък обхват на стойности и не може например да запише стойността **`1e-50`**. Така, при конвертиране на числа с плаваща запетая към **`decimal`**, може да се получат грешки от препълване.

Символни данни (стрингове)

Символните (текстовите) данни в компютърната техника представляват текст, кодиран чрез поредици от байтове. Има различни схеми за кодиране на текстови данни. Повечето от тях кодират един символ с един байт или с поредица от няколко байта. Такива са кодиращите схеми ASCII, Windows-1251, UTF-8 и UTF-16.

Кодиращи схеми (encodings)

Кодиращата схема **ASCII** съпоставя уникален номер на буквите от латинската азбука и някои други символи и специални знаци и ги записва в един байт. ASCII стандартът съдържа общо 127 символа, всеки от които се записва в 1 байт. Текст, записан като поредица от байтове по стандарта ASCII не може да съдържа кирилица и символи от други азбуки като арабската, корейската и китайската.

По подобие на ASCII стандарта кодиращата схема **Windows-1251** съпоставя на буквите от латинската азбука, кирилицата и някои други символи и специални знаци по един байт. Кодирането Windows-1251 дефинира номера на общо 256 символа – точно колкото са различните стойности, които могат да се запишат с един байт. Текст, записан по стандарта

Windows-1251 може да съдържа кирилица и латиница, но не може да съдържа арабски, индийски и китайски символи.

Кодирането **UTF-8** е съвсем различно. В него могат да бъдат записани всички символи от Unicode стандарта – буквите и знаците, използвани във всички масово разпространени езици по света (кирилица, латиница, арабски, китайски, японски, корейски и много други азбуки и писмени системи). Кодирането UTF-8 съдържа над половин милион символа. При UTF-8 по-често използваните символи се кодират в 1 байт (например латиницата), по-рядко използваните символи се кодират в 2 байта (например кирилицата) и още по-рядко използваните символи се кодират с 3 или 4 байта (например китайската, японската и корейската азбука).

Кодирането **UTF-16**, подобно на UTF-8 може да представи текстове от всички по-масово използвани по света езици и писмени системи, описани в Unicode стандарта. При UTF-16 всеки символ се записва в 16 бита, т.е. в 2 байта, а някои по-рядко използвани символи се представят като поредица от две 16-битови стойности.

Представяне на поредици от символи

Поредиците от символи могат да се представят по-няколко начина. Най-разпространеният начин за записване на текст в паметта на компютъра е в 2 или 4 байта да се запише дължината на текста, следван от поредица от байтове, които представят самия текст в някакво кодиране (например Windows-1251 или UTF-8).

Друг, по-малко разпространен начин за записване на текстове в паметта, типичен за езика C, представя текстовете чрез поредица от символи, най-често в еднобайтово кодиране, следвани от специален завършващ символ, най-често с код 0. Така дължината на текста, записан на дадена позиция в паметта, не е предварително известна, което се счита за сериозен недостатък в много ситуации.

Типът char

Типът **`char`** в езика C# представлява 16-битова стойност, в която е кодиран един Unicode символ или част от Unicode символ. При повечето азбуки (например използваните от всички европейски езици) една буква се записва в една 16-битова стойност и по тази причина обикновено се счита, че една променлива от тип **`char`** представя един символ. Ето един пример:

```
char ch = 'A';
Console.WriteLine(ch);
```

Типът string

Типът **`string`** в C# съдържа текст, кодиран в UTF-16. Един стринг в C# се състои от 4 байта дължина и поредица от символи, записани като 16-битови стойности от тип **`char`**. В типа **`string`** може да се запишат текстове на всички широко разпространени азбуки и писмени системи от човешките езици – латиница, кирилица, китайски, японски, арабски и много, много други. Ето един пример за използване на типа **`string`**:

```
string str = "Example";
Console.WriteLine(str);
```

Упражнения

1. Превърнете числата 151, 35, 43, 251 и -0,41 в двоична бройна система.
2. Превърнете числото $\mathbf{1111010110011110_{(2)}}$ в шестнадесетична и в десетична бройна система.
3. Превърнете шестнайсетичните числа 2A3E, FA, FFFF, 5A0E9 в двоична и десетична бройна система.
4. Да се напише програма, която преобразува десетично число в двоично.
5. Да се напише програма, която преобразува двоично число в десетично.
6. Да се напише програма, която преобразува десетично число в шестнадесетично.
7. Да се напише програма, която преобразува шестнадесетично число в десетично.
8. Да се напише програма, която преобразува шестнадесетично число в двоично.
9. Да се напише програма, която преобразува двоично число в шестнадесетично.
10. Да се напише програма, която преобразува двоично число в десетично по схемата на Хорнер.
11. Да се напише програма, която преобразува римските числа в арабски.
12. Да се напише програма, която преобразува арабските числа в римски.
13. Да се напише програма, която по зададени **`N`**, **`S`**, **`D`** (**`2 ≤ S, D ≥ 16`**) преобразува числото **`N`** от бройна система с основа **`S`** към бройна система с основа **`D`**.
14. Да се напише програма, която по дадено цяло число извежда на конзолата двоичното представяне на числото.
15. Опитайте да сумирате 50 000 000 пъти числото 0.000001. Използвайте цикъл и събиране (не директно умножение). Опитайте с типовете **`float`** и **`double`** и след това с **`decimal`**. Забелязвате ли разликата в резултатите и в скоростта?

16. * Да се напише програма, която отпечатва стойността на мантисата, знака на мантисата и стойността на експонентата за числа тип `float` (32-битови числа с плаваща запетая съгласно стандарта IEEE 754). Пример: за числото -27,25 да се отпечата: знак = 1, експонента = 10000011, мантиса = 10110100000000000000000.

Решения и упътвания

1. Използвайте методите за превръщане от една бройна система в друга. Можете да сверите резултатите си с калкулатора на Windows, който поддържа работа с бройни системи след превключване в режим "Scientific".
2. Погледнете упътването за предходната задача.
3. Погледнете упътването за предходната задача.
4. Правилото е "делим на 2 и долепяме остатъците в обратен ред". За делене с остатък използваме оператора `%`.
5. Започнете от сума 0. Умножете най-десния бит с 1 и го прибавете към сумата. Следващия бит вляво умножете по 2 и добавете към сумата. Следващия бит отляво умножете по 4 и добавете към сумата и т.н.
6. Правилото е "делим на основата на системата (16) и долепяме остатъците в обратен ред". Трябва да си напишем логика за отпечатване на шестнайсетична цифра по дадена стойност между 0 и 15.
7. Започнете от сума 0. Умножете най-дясната цифра с 1 и я прибавете към сумата. Следващата цифра вляво умножете по 16 и я добавете към сумата. Следващата цифра вляво умножете по 16*16 и я добавете към сумата и т.н. до най-лявата шестнайсетична цифра.
8. Ползвайте бързия начин за преминаване между шестнайсетична и двоична бройна система (всяка шестнайсетична цифра съответства на 4 двоични бита).
9. Ползвайте бързия начин за преминаване между двоична и шестнайсетична бройна система (всяка шестнайсетична цифра съответства на 4 двоични бита).
10. Приложете директно схемата на Хорнер.
11. Сканирайте цифрите на римското число отляво надясно и ги добавяйте към сума, която първоначално е инициализирана с 0. При обработката на всяка римска цифра я взимайте с положителен или отрицателен знак в зависимост от следващата цифра (дали има по-малка или по-голяма десетична стойност).
12. Разгледайте съответствията на числата от 1 до 9 с тяхното римско представяне с цифрите "I", "V" и "X":

    1 -> I
    2 -> II

3 -> III
4 -> IV
5 -> V
6 -> VI
7 -> VII
8 -> VIII
9 -> IX

Имаме абсолютно аналогични съответствия на числата 10, 20, ..., 90 с тяхното представяне с римските цифри "X", "L" и "C", нали? Имаме аналогични съответствия между числата 100, 200, ..., 900 и тяхното представяне с римските цифри "C", "D" и "M" и т.н.

Сега сме готови да преобразуваме числото N в римска бройна система. То трябва да е в интервала [1...3999], иначе съобщаваме за грешка. Първо отделяме хилядите (N / 1000) и ги заместваме с римския им еквивалент. След това отделяме стотиците ((N / 100) % 10) и ги заместваме с римския им еквивалент и т.н.

13. Можете да прехвърлите числото от бройна система с онова **`S`** към бройна система с онова 10, а после от бройна система с основа 10 към бройна система с онова **`D`**.

14. За задачата можете да използвате резултата от операцията **`%2`** (остатък при деление с 2).

15. Ако изпълните правилно изчисленията, ще получите съответно 32 (за **`float`**), 49.9999999657788 (за **`double`**) и 50 (за **`decimal`**). Като скорост ще установите, че събирането на **`decimal`** стойности е поне 10 пъти по-бавно от събирането на **`double`** стойности.

16. Използвайте специалният метод за представяне на числа с плаваща запетая с двойна точност като 64 битово цяло число **`System.BitConverter.DoubleToInt64Bits(<double>)`**, след което използвайте подходящи побитови операции (измествания и битови маски).

Telerik Academy

Присъедини се към Академията на Телерик!

АКАДЕМИЯТА НА ТЕЛЕРИК предоставя безплатно практическо обучение, насочено към всички млади хора, желаещи да станат умели .NET софтуерни инженери и да се присъединят към екипа на Телерик.

Академията подготвя бъдещите софтуерни специалисти за сложността, разнообразието и динамиката на днешната ИТ действителност, като за целта всички курсисти се обучават да работят с най-новите технологии на Майкрософт и черпят знания и опит от доказали се в областта на програмирането лектори и разработчици на софтуер.

В академията ще получите задълбочени знания и опит, изучавайки:

C#, .NET технологии, бази данни, SQL, Silverlight, WPF, ASP.NET, HTML5, разработка на мобилни приложения за iOS, Android и Windows Phone, основите на софтуерното инженерство

Академията на Телерик ви дава възможност да:

- **Учите напълно БЕЗПЛАТНО**
- **Изберете сред редица РАЗЛИЧНИ КУРСОВЕ**
- **Овладеете ОСНОВИТЕ на софтуерното инженерство**
- **Усвоите ПРОЦЕСА за разработка на софтуер**
- **Получите задълбочени теоретични и практически ИТ ПОЗНАНИЯ**
- **Станете умел .NET СОФТУЕРЕН ИНЖЕНЕР**
- **Започнете своята ИТ кариера в ТЕЛЕРИК – РАБОТОДАТЕЛ #1 в България за 2010 г.**

Само в рамките на две години **АКАДЕМИЯТА НА ТЕЛЕРИК** за софтуерни инженери успя да се наложи като безспорен лидер у нас в предлагането на допълнително обучение за софтуерни специалисти, спомагайки за успешния старт в кариерното развитие на стотици ентусиазирани младежи.

academy.telerik.com

academy@telerik.com *facebook.com/TelerikAcademy*

telerik
deliver more than expected

Глава 9. Методи

В тази тема...

В настоящата тема ще се запознаем подробно с това какво е метод и защо трябва да използваме методи. Ще разберем как се декларират методи, какво е сигнатура на метод, как се извикват методи, как им се подават параметри и как методите връщат стойност. След като приключим темата, ще знаем как да създадем собствен метод и съответно как да го използваме (извикваме) в последствие. Накрая ще препоръчаме някои утвърдени практики при работата с методи.

Всичко това ще бъде подкрепено с подробно обяснени примери и допълнителни задачи, с които читателят ще може да упражни наученото.

Подпрограмите в програмирането

В ежедневието ни, при решаването на даден проблем, особено, ако е по-сложен, прилагаме принципа на древните римляни "разделяй и владей". Съгласно този принцип, проблемът, който трябва да решим, се разделя на множество по-малки подпроблеми. Самостоятелно разгледани, те са по-ясно дефинирани и по-лесно решими, в сравнение с търсенето на решение на изходния проблем като едно цяло. Накрая, от решенията на всички подпроблеми, създаваме решението на цялостния проблем.

По същата аналогия, когато пишем дадена програма, целта ни е с нея да решим конкретна задача. За да го направим ефективно и да улесним работата си, прилагаме принципа "разделяй и владей". Разбиваме поставената ни задача на подзадачи, разработваме решения на тези подзадачи и накрая ги "сглобяваме" в една програма. Решенията на тези подзадачи наричаме подпрограми (subroutines).

В някои езици за програмиране подпрограмите могат да се срещнат под наименованията функции (functions) или процедури (procedures). В C#, те се наричат методи (methods).

Какво е "метод"?

Метод (method) е съставна част от програмата, която решава даден проблем, може да приема параметри и да връща стойност.

В методите се извършва цялата обработка на данни, която програмата трябва да направи, за да реши поставената задача. Методите съдържат логиката на програмата и те са мястото, където се извършва реалната работа. Затова можем да ги приемем като строителен блок на програмата. Съответно, имайки множество от простички блокчета – отделни методи, можем да създаваме големи програми, с които да решим по-сложни проблеми. Ето например как изглежда един метод за намиране лице на правоъгълник:

```
static double GetRectangleArea(double width, double height)
{
  double area = width * height;
  return area;
}
```

Защо да използваме методи?

Има много причини, които ни карат да използваме методи. Ще разгледаме някои от тях и с времето ще се убедите, че методите са нещо, без което не можем, ако искаме да програмираме сериозно.

По-добро структуриране и по-добра четимост

При създаването на една програма, е добра практика да използваме методи, за да я направим добре структурирана и лесно четима не само за нас, но и за други хора.

Довод за това е, че за времето, през което съществува една програма, средно само 20% от усилията, които се заделят за нея, се състоят в създаване и тестване на кода. Останалата част е за поддръжка и добавяне на нови функционалности към началната версия. В повечето случаи, след като веднъж кодът е написан, той не се поддържа и модифицира само от създателя му, но и от други програмисти. Затова е важно той да е добре структуриран и лесно четим.

Избягване на повторението на код

Друга много важна причина, заради която е добре да използваме методи е, че по този начин избягваме повторението на код. Това е пряко свързано с концепцията за преизползване на кода.

Преизползване на кода

Добър стил на програмиране е, когато използваме даден фрагмент програмен код повече от един или два пъти в програмата си, да го дефинираме като отделен метод, за да можем да го изпълняваме многократно. По този начин освен, че избягваме повторението на код, програмата ни става по-четима и по-добре структурирана.

Повтарящият се код е вреден и доста опасен, защото силно затруднява поддръжката на програмата и води до грешки. При промяната на повтарящ се код често пъти програмистът прави промени само на едно място, а останалите повторения на кода си остават същите. Така например, ако е намерен дефект във фрагмент от 50 реда код, който се повтаря на 10 места в програмата, за да се поправи дефектът, трябва на всичките тези 10 места да се преправи кода по един и същ начин. Това най-често не се случва поради невнимание и програмистът обикновено поправя само някои от повтарящите се дефекти, но не всички. Например в нашия случай е възможно програмистът да поправи проблема на 8 от 10-те места, в които се повтаря некоректния код и това в крайна сметка ще доведе до некоректно поведение на програмата в някои случаи, което е трудно да се установи и поправи.

Деклариране, имплементация и извикване на собствен метод

Преди да продължим по-нататък, ще направим разграничение между три действия свързани със съществуването на един метод – деклариране, имплементация (създаване) и извикване на метод.

Деклариране на метод наричаме регистрирането на метода в програмата, за да бъде разпознаван в останалата част от нея.

Имплементация (създаване) на метода, е реалното написване на кода, който решава конкретната задача, която методът решава. Този код се съдържа в самия метод и реализира неговата логика.

Извикване е процесът на стартиране на изпълнението, на вече декларирания и създаден метод, от друго място на програмата, където трябва да се реши проблемът, за който е създаден извикваният метод.

Деклариране на собствен метод

Преди да се запознаем как можем да декларираме метод, трябва да знаем къде е позволено да го направим.

Къде е позволено да декларираме метод

Въпреки, че формално все още не сме запознати как се декларира клас, от примерите, които сме разглеждали до сега в предходните глави, знаем, че всеки клас има отваряща и затваряща фигурни скоби – "`{`" и "`}`", между които пишем програмния код. Повече подробности за това, ще научим в главата "Дефиниране на класове", но го споменаваме тук, тъй като един метод може да съществува само ако е деклариран **между** отварящата и затварящата скоби на даден клас – "`{`" и "`}`". Допълнително изискване е методът, трябва да бъде деклариран **извън** имплементацията на друг метод (за това малко по-късно).

⚠	**В езика C# можем да декларираме метод единствено в рамките на даден клас – между отварящата "{" и затварящата "}" му скоби.**

Най-очевидният пример за методи е вече познатият ни метод **`Main(…)`** – винаги го декларираме между отварящата и затварящата скоба на нашия клас, нали? Да си припомним това с един пример:

HelloCSharp.cs

```
public class HelloCSharp
{ // Opening brace of the class

   // Declaring our method between the class' braces
   public static void Main(string[] args)
   {
      Console.WriteLine("Hello C#!");
   }
} // Closing brace of the class
```

Декларация на метод

Декларирането на метод, представлява **регистриране** на метода в нашата програма. То става чрез следната декларация:

```
[public] [static] <return_type> <method_name>([<param_list>])
```

Задължителните елементи в декларацията на един метод са:

- Тип на връщаната от метода стойност – **`<return_type>`**.
- Име на метода – **`<method_name>`**.
- Списък с параметри на метода – **`<param_list>`** – може да е празен списък или да съдържа поредица от декларации на параметри.

За онагледяване на елементите от декларацията на методите, можем да погледнем **`Main(…)`** метода в примера **`HelloCSharp`** от предходната секция:

```
public static void Main(string[] args)
```

При него, типът на връщаната стойност е **`void`** (т.е. методът не връща резултат), името му е **`Main`**, следвано от кръгли скоби, в които има списък с параметри, състоящ се от един параметър – масивът **`string[] args`**.

Последователността, в която трябва да се поставят отделните елементи от декларацията на метода е строго определена. Винаги на първо място е типът на връщаната стойност **`<return_type>`**, следвана от името на метода **`<method_name>`** и накрая, списък с параметри **`<param_list>`** ограден с кръгли скоби – "**`(`**" и "**`)`**". Опционално в началото на декларацията може да има модификатори за достъп (например **`public`** и **`static`**).

⚠	**При деклариране на метод, спазвайте последователността, в която се описват основните му елементи: първо тип на връщана стойност, след това име на метода и накрая списък от параметри, ограден с кръгли скоби.**

Списъкът от параметри може да е празен и тогава просто пишем "**`()`**" след името на метода. Дори методът да няма параметри, кръглите скоби трябва да присъстват задължително в декларацията му.

⚠	**Кръглите скоби – "(" и ")", винаги следват името на метода, независимо дали той е с или без параметри.**

За момента, ще пропуснем разглеждането какво е **`<return_type>`** и ще приемем, че на това място трябва да стои ключовата дума **`void`**, която указва, че методът не връща никаква стойност. По-късно ще обясним какво друго можем да поставим на нейно място.

Ключовите думи `public` и `static` в описанието на декларацията по-горе са незадължителни и имат специално предназначение, което ще разгледаме по-късно в тази глава. За момента ще разглеждаме методи, които винаги имат `static` в декларацията си. Повече за методите, които не са декларирани като `static`, ще научим от главата "Дефиниране на класове".

Сигнатура на метод

Преди да продължим с основните елементи от декларацията на метода, трябва да обърнем внимание на нещо много важно. В обектно-ориентираното програмиране, начинът, по който еднозначно се идентифицира един метод е чрез двойката елементи от декларацията му – име на метода и списък от неговите параметри. Тези два елемента определят така наречената **спецификация на метода** (често в литературата се среща и като **сигнатура на метода**).

C#, като език за обектно-ориентирано програмиране, също разпознава еднозначно различните методи, използвайки тяхната спецификация (сигнатура) – името на метода `<method_name>` и списъкът с параметрите му – `<param_list>`.

Трябва да обърнем внимание, че типът на връщаната стойност на един метод е част от декларацията му, но не е част от сигнатурата му.

Това, което идентифицира един метод, е неговата сигнатура. Връщаният тип не е част от нея. Причината е, че ако два метода се различават само по връщания тип, то не може еднозначно да се идентифицира кой от тях трябва да бъде извикан.

По-подробен пример, защо типът на връщаната стойност не е част от сигнатурата на метода ще разгледаме по-късно в тази глава.

Име на метод

Всеки метод, решава някаква подзадача от цялостния проблем, с който се занимава програмата ни. Името на метода се използва при извикването му. Когато извикаме (стартираме) даден метод, ние изписваме името му и евентуално подаваме стойности на параметрите му (ако има такива).

В примера показан по-долу, името на метода е `PrintLogo`:

```
static void PrintLogo()
{
  Console.WriteLine("Microsoft");
  Console.WriteLine("www.microsoft.com");
}
```

Правила за именуване на метод

Добре е, когато декларираме името на метода, да спазваме правилата за именуване на методи, препоръчани ни от Microsoft:

- Името на методите трябва да започва с главна буква.
- Трябва да се прилага правилото **`PascalCase`**, т.е. всяка нова дума, която се долепя като част от името на метода, трябва да започва с главна буква.
- Имената на методите е препоръчително да бъдат съставени от глагол или от глагол и съществително име.

Нека отбележим, че тези правила не са задължителни, а препоръчителни. Но принципно, ако искаме нашият C# код да следва стила на всички добри програмисти по света, е най-добре да спазваме конвенциите на Microsoft.

Ето няколко примера за добре именувани методи:

```
Print
GetName
PlayMusic
SetUserName
```

Ето няколко примера за лошо именувани методи:

```
Abc11
Yellow___Black
foo
_Bar
```

Изключително е важно името на метода трябва да описва неговата цел. Идеята е, ако човек, който не е запознат с програмата ни, прочете името на метода, да добие представа какво прави този метод, без да се налага да разглежда кода му.

При определяне на името на метод се препоръчва да се спазват следните правила:

- **Името на метода трябва да описва неговата цел.**
- **Името на метода трябва да започва с главна буква.**
- **Трябва да се прилага правилото PascalCase.**
- **Името на метода трябва да е съставено от глагол или от двойка - глагол и съществително име.**

Модификатори (modifiers)

Модификатор (modifier) наричаме ключова дума в езика C#, която дава допълнителна информация на компилатора за даден код.

Модификаторите, които срещнахме до момента са **`public`** и **`static`**. Сега ще опишем на кратко какво представляват те. Детайлно обяснение за тях, ще бъде дадено по-късно в главата "Дефиниране на класове". Да започнем с един пример:

```
public static void PrintLogo()
{
   Console.WriteLine("Microsoft");
   Console.WriteLine("www.microsoft.com");
}
```

В примера декларираме публичен метод чрез модификатора **`public`**. Той е специален вид модификатор, наречен **модификатор за достъп (access modifier)** и се използва, за да укаже, че извикването на метода може да става от кой да е C# клас, независимо къде се намира той. Публичните методи нямат ограничение кой може да ги извиква.

Друг пример за модификатор за достъп, който може да срещнем, е модификаторът **`private`**. Като предназначение, той е противоположен на **`public`**, т.е. ако един метод бъде деклариран с модификатор за достъп **`private`**, то този метод не може да бъде извикан извън класа, в който е деклариран.

Когато един метод няма дефиниран модификатор за достъп (например **`public`** или **`private`**), той е достъпен от всички класове в текущото асембли, но не и от други асемблита (например от други проекти във Visual Studio). По тази причина за малки програмки, каквито са повечето примери в настоящата глава, няма да задаваме модификатори за достъп.

За момента, единственото, което трябва да научим е, че в декларацията си един метод може да има не повече от един модификатор за достъп.

Когато един метод притежава ключовата дума **`static`**, в декларацията си, наричаме метода **статичен**. За да бъде извикан един статичен метод, няма нужда да бъде създадена инстанция на класа, в който той е деклариран. За момента приемете, че методите които пишем, трябва да са статични, а работата с нестатични методи ще разгледаме по-късно в главата "Дефиниране на класове".

Имплементация (създаване) на собствен метод

След като декларираме метода, следва да напишем неговата имплементация. Както обяснихме по-горе, **имплементацията (тялото)** на метода се състои от кода, който ще бъде изпълнен при извикването на метода. Този код трябва да бъде поставен в тялото на метода и той реализира неговата логика.

Тяло на метод

Тяло на метод наричаме програмният код, който се намира между фигурните скоби "**{**" и "**}**", следващи непосредствено декларацията на метода.

```
static <return_type> <method_name>(<parameters_list>)
{
  // ... code goes here – in the method’s body ...
}
```

Реалната работа, която методът извършва, се намира именно в тялото на метода. В него трябва да бъде описан алгоритъмът, по който методът решава поставения проблем.

Примери за тяло на метод сме виждали много пъти, но сега ще изложим още един:

```
static void PrintLogo()
{ // Method’s body start here
  Console.WriteLine("Microsoft");
  Console.WriteLine("www.microsoft.com");
} // ... And finishes here
```

Обръщане отново внимание на едно от правилата, къде може да се декларира метод:

⚠	**Метод НЕ може да бъде деклариран в тялото на друг метод.**

Локални променливи

Когато декларираме променлива в тялото на един метод, я наричаме **локална променлива (local variable)** за метода. Когато именуваме една променлива трябва да спазваме правилата за идентификатори в C# (вж. глава "Примитивни типове и променливи").

Областта, в която съществува и може да се използва една локална променлива, започва от реда, на който сме я декларирали и стига до затварящата фигурна скоба "**}**" на тялото на метода. Тази област се нарича **област на видимост** на променливата. Ако след като сме декларирали една променлива се опитаме да декларираме в същия метод друга променлива със същото име, ще получим грешка при компилация. Например да разгледаме следния код:

```
static void Main()
{
  int x = 3;
  int x = 4;
}
```

Компилаторът няма да ни позволи да ползваме името **`x`** за две различни променливи и ще изведе със съобщение подобно на следното:

```
A local variable named 'x' is already defined in this scope.
```

Програмен блок (block) наричаме код, който се намира между отваряща и затваряща фигурни скоби "**{**" и "**}**".

Ако декларираме променлива в блок, тя отново се нарича локална променлива, и областта й на съществуване е от реда, на който бъде декларирана, до затварящата скоба на блока, в който се намира.

Извикване на метод

Извикване на метод наричаме стартирането на изпълнението на кода, който се намира в тялото на метода.

Извикването на метода става просто като напишем името на метода **`<method_name>`**, следвано от кръглите скоби и накрая сложим знака за край на ред – ";":

```
<method_name>();
```

По-късно ще разгледаме и случая, когато извикваме метод, който има списък с параметри.

За да имаме ясна представа за извикването, ще покажем как бихме извикали метода, който използвахме в примерите по-горе – **`PrintLogo()`**:

```
PrintLogo();
```

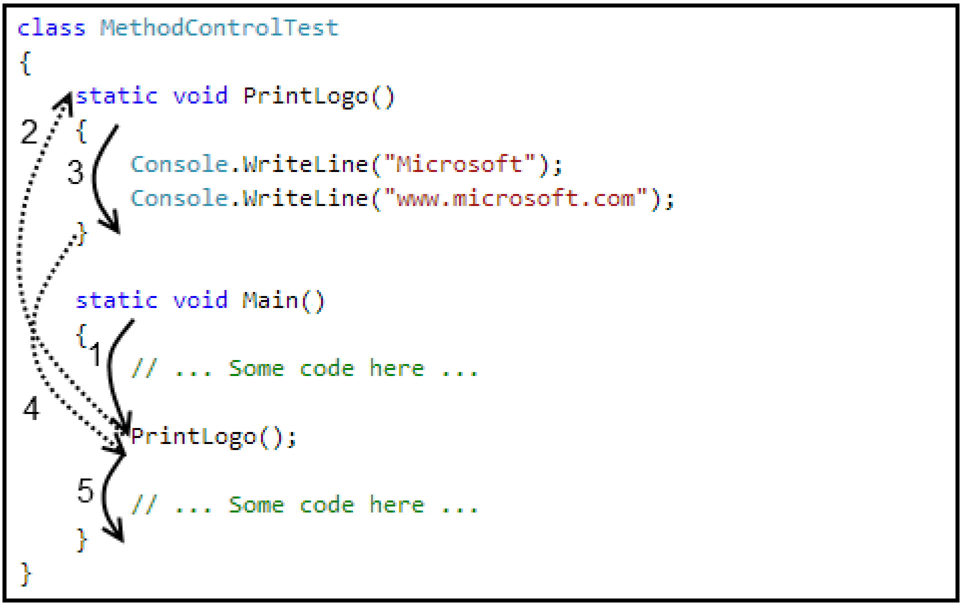
Изходът от изпълнението на метода ще бъде:

```
Microsoft
www.microsoft.com
```

Предаване на контрола на програмата при извикване на метод

Когато изпълняваме един метод, той притежава контрола над програмата. Ако в тялото му обаче, извикаме друг метод, то тогава извикващият метод ще предаде контрола на извиквания метод. След като извикваният метод приключи изпълнението си, той ще върне контрола на метода, който го е извикал. Изпълнението на първия метод ще продължи от следващия ред.

Например, нека от метода **`Main()`** извикаме метода **`PrintLogo()`**:



```
class MethodControlTest
{
    static void PrintLogo()
    {
        Console.WriteLine("Microsoft");
        Console.WriteLine("www.microsoft.com");
    }

    static void Main()
    {
        // ... Some code here ...

        PrintLogo();

        // ... Some code here ...
    }
}
```

Първо ще се изпълни кодът от метода **`Main()`**, който е означен с (1), след това контролът на програмата ще се предаде на метода **`PrintLogo()`** – пунктираната стрелка (2). След това, ще се изпълни кодът в метода **`PrintLogo()`**, номериран с (3). След приключване на работата на метода **`PrintLogo()`** управлението на програмата ще бъде върнато обратно на метода **`Main()`** – пунктираната стрелка (4). Изпълнението на метода **`Main()`** ще продължи от реда, който следва извикването на метода **`PrintLogo()`** – стрелката маркирана с (5).

От къде може да извикаме метод?

Един метод може да бъде извикван от следните места:

- От главния метод на програмата – **`Main()`**:

```
static void Main()
{
   PrintLogo();
}
```

- От някой друг метод, например:

```
static void PrintLogo()
{
   Console.WriteLine("Microsoft");
   Console.WriteLine("www.microsoft.com");
}

static void PrintCompanyInformation()
```

```
{
   // Invoking the PrintLogo() method
   PrintLogo();

   Console.WriteLine("Address: One, Microsoft Way");
}
```

- Методът може да бъде извикан от собственото си тяло. Това се нарича **рекурсия (recursion)**, но ще се запознаем нея по-подробно в следващата глава – "Рекурсия".

Независимост между декларацията и извикването на метод

Когато пишем на C# наредбата на методите в класовете не е от значение и е позволено извикването на метод да предхожда неговата декларация и имплементация. За да онагледим това, нека разгледаме следния пример:

```
static void Main()
{
   // ...
   PrintLogo();
   // ...
}

static void PrintLogo()
{
   Console.WriteLine("Microsoft");
   Console.WriteLine("www.microsoft.com");
}
```

Ако създадем клас, който съдържа горния код, ще се убедим, че независимо че извикването на метода е на по-горен ред от декларацията на метода, програмата ще се компилира и изпълни без никакъв проблем. В някои други езици за програмиране, като например Паскал, извикването на метод, който е дефиниран по-надолу от мястото на извикването му, не е позволено.

Ако един метод бива извикван в същия клас, където е деклариран и имплементиран, то той може да бъде извикан на ред по-горен от реда на декларацията му.

Използване на параметри в методите

Много често, за да реши даден проблем, методът се нуждае от допълнителна информация, която зависи от контекста, в който той се изпълнява.

Например, ако имаме метод, който намира лице на квадрат, в тялото му е описан алгоритъма, по който се намира лицето (формулата $S = a^2$). Тъй като лицето на квадрата зависи от дължината на неговата страна, при пресмятането на лицето на всеки отделен квадрат, методът ни ще се нуждае от стойност, която задава дължината на страната му. Затова, ние трябва да му я подадем някак и за тази цел се използват параметрите.

Деклариране на метод

За да можем да подадем информация на даден метод, която е нужна за неговата работа, използваме **списък от параметри**. Този списък, поставяме между кръглите скоби в декларацията на метода, след името му:

```
static <return_type> <method_name>(<parameters_list>)
{
  // Method's body
}
```

Списъкът от параметри `<parameters_list>`, представлява списък от нула или повече **декларации на променливи**, разделени със запетая, които ще бъдат използвани в процеса на работа на метода:

```
<parameters_list> = [<type_1> <name_1>[, <type_i> <name_i>]],
където i = 2, 3,...
```

Когато създаваме метода и ни трябва дадена информация за реализирането на алгоритъма, избираме тази променлива от списъка от параметри, чийто тип е `<type_i>` и я използваме съответно чрез името й `<name_i>`.

Типът на параметрите в списъка може да бъде различен. Той може да бъде както примитивни типове – `int`, `double`, ... така и обекти (например `string` или масиви – `int[]`, `double[]`, `string[]`, ...).

Метод за извеждане на фирмено лого – пример

За да добием по-ясна представа, нека модифицираме примера, който извежда логото на компанията "Microsoft" по следния начин:

```
static void PrintLogo(string logo)
{
  Console.WriteLine(logo);
}
```

По този начин, нашият метод вече няма да извежда само "Microsoft" като резултат от изпълнението си, но логото на всяка компания, чието име подадем като параметър от тип `string`. В примера виждаме също как използваме информацията подадена ни в списъка от параметри – променливата `logo`, дефинирана в списъка от параметри, се използва в тялото на метода чрез името, с което сме я дефинирали.

Метод за сумиране цените на книги в книжарница – пример

По-горе казахме, че когато е нужно, можем да подаваме като параметри на метода и масиви – **`int[]`**, **`double[]`**, **`string[]`**, … Нека в тази връзка разгледаме друг пример.

Ако сме в книжарница и искаме да пресметнем сумата, която дължим за всички книги, които желаем да закупим, можем да си създадем метод, който приема като входни данни цените на отделните книги във вид масив от тип **`decimal[]`** и връща общата им стойност, която трябва да заплатим на продавача:

```
static void PrintTotalAmountForBooks(decimal[] prices)
{
   decimal totalAmount = 0;
   foreach (decimal singleBookPrice in prices)
   {
      totalAmount += singleBookPrice;
   }
   Console.WriteLine("The total amount of all books is:" +
      totalAmount);
}
```

Поведение на метода в зависимост от входните данни

Когато декларираме метод с параметри, целта ни е всеки път, когато извикваме този метод, работата му да се променя в зависимост от входните данни. С други думи, алгоритъмът, който ще опишем в метода, ще бъде един, но крайният резултат ще бъде различен, в зависимост от това какви входни данни сме подали на метода чрез стойностите на входните му параметри.

> **Когато даден метод приема параметри, поведението му, зависи от тях.**

Метод за извеждане знака на едно число – пример

За да стане ясно как поведението (изпълнението) на метода зависи от входните параметри, нека разгледаме следния метод, на който подаваме едно цяло число (от тип **`int`**), и в зависимост от това, дали числото е положително, отрицателно или нула, съответно той извежда на конзолата стойност "**`Positive`**", "**`Negative`**" или "**`Zero`**":

```
static void PrintSign(int number)
{
   if (number > 0)
   {
      Console.WriteLine("Positive");
   }
```

```
    else if (number < 0)
    {
        Console.WriteLine("Negative");
    }
    else
    {
        Console.WriteLine("Zero");
    }
}
```

Методи с няколко параметъра

До сега разглеждахме примери, в които методите имат списък от параметри, който се състои от един единствен параметър. Когато декларираме метод обаче, той може да има толкова параметри, колкото са му необходими.

Например, когато търсим по-голямото от две числа, ние подаваме два параметъра:

```
static void PrintMax(float number1, float number2)
{
    float max = number1;
    if (number2 > number1)
    {
        max = number2;
    }
    Console.WriteLine("Maximal number: " + max);
}
```

Особеност при декларацията на списък с много параметри

Когато в списъка с параметри декларираме повече от един параметър от един и същ тип, трябва да знаем, че не можем да използваме съкратения запис за деклариране на променливи от един и същи тип, както е позволено в самото тяло на метода, т.е. следният списък от параметри е невалиден:

```
float var1, var2;
```

Винаги трябва да указваме типа на параметъра в списъка с параметри на метода, независимо че някой от съседните му параметри е от същия тип.

Например, тази декларация на метод е неправилна:

```
static void PrintMax(float var1, var2)
```

Съответно, същата декларация, изписана правилно, е:

```
static void PrintMax(float var1, float var2)
```

Извикване на метод с параметри

Извикването на метод с един или няколко параметъра става по същия начин, по който извиквахме метод без параметри. Разликата е, че между кръглите скоби, след името на метода, поставяме стойности. Тези стойности ще бъдат присвоени на съответните параметри от декларацията на метода и при изпълнението си, методът ще работи с тях.

Ето няколко примера за извикване на методи с параметри:

```
PrintSign(-5);
PrintSign(balance);

PrintMax(100, 200);
```

Разлика между параметри и аргументи на метод

Преди да продължим, трябва да направим едно разграничение между наименованията на параметрите в списъка от параметри в декларацията на метода и стойностите, които подаваме при извикването на метода.

За по-голяма яснота, при декларирането на метода, елементите на списъка от параметрите му, ще наричаме **параметри** (някъде в литературата могат да се срещнат също като "**формални параметри**").

По време на извикване на метода, **стойностите**, които подаваме на метода, наричаме **аргументи** (някъде могат да се срещнат под понятието "**фактически параметри**").

С други думи, елементите на списъка от параметри **`var1`** и **`var2`**, наричаме параметри:

```
static void PrintMax(float var1, float var2)
```

Съответно стойностите, при извикването на метода **`-23.5`** и **`100`**, наричаме аргументи:

```
PrintMax(100, -23.5);
```

Подаване на аргументи от примитивен тип

Както току-що научихме, когато в C# подадем като аргумент на метод дадена променлива, стойността й се **копира** в параметъра от декларацията на метода. След това, копието ще бъде използвано в тялото на метода.

Има, обаче, една особеност: когато съответният параметър от декларацията на метода е от **примитивен тип**, това практически не оказва никакво влияние на подадената като аргумент променлива в кода след извикването на метода.

Например, ако имаме следния метод:

```
static void PrintNumber(int numberParam)
{
   // Modifying the primitive-type parameter
   numberParam = 5;

   Console.WriteLine("in PrintNumber() method, after the " +
      "modification, numberParam is {0}: ", numberParam);
}
```

Извиквайки го от метода **`Main()`**:

```
static void Main()
{
   int numberArg = 3;

   // Copying the value 3 of the argument numberArg to the
   // parameter numberParam
   PrintNumber(numberArg);

   Console.WriteLine("in the Main() method number is: " +
      numberArg);
}
```

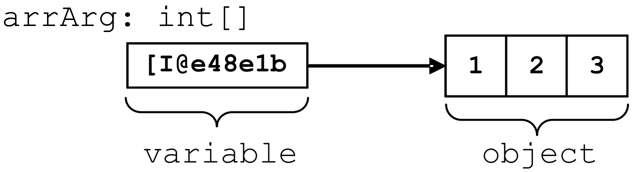
Стойността **`3`** на променливата **`numberArg`**, се копира в параметъра **`numberParam`**. След като бъде извикан методът **`PrintNumber()`**, на параметъра **`numberParam`** се присвоява стойността **`5`**. Това не рефлектира върху стойността на променливата **`numberArg`**, тъй като при извикването на метода, в променливата **`numberParam`** се пази **копие** на стойността на подадения аргумент. Затова, методът **`PrintNumber()`** отпечатва числото **`5`**. Съответно, след извикването на метода **`PrintNumber()`**, в метода **`Main()`** отпечатваме стойността на променливата **`numberArg`** и виждаме, че тя не е променена. Ето и изходът от изпълнението на горната програма:

```
in PrintNumber() method, after the modification numberParam is:5
in the Main() method number is: 3
```

Подаване на аргументи от референтен тип

Когато трябва да декларираме (и съответно извикаме) метод, чийто параметри са от **референтен тип** (например масиви), трябва да бъдем много внимателни.

Преди да обясним защо, нека припомним нещо от главата "Масиви". Масивът, като всеки референтен тип, се състои от променлива (референция) и стойност – реалната информация в паметта на компютъра (нека я наречем **обект**). Съответно в нашия случай обектът представлява реалният масив от елементи. Променливата пази адреса на обекта в паметта (т.е. мястото в паметта, където се намират елементите на масива):



Когато оперираме с масиви, винаги го правим чрез променливата, с която сме ги декларирали. Така е и с всеки референтен тип. Следователно, когато подаваме аргумент от референтен тип, **стойността**, която е записана в променливата-аргумент, се **копира** в променливата, която е параметър в списъка от параметри на метода. Но какво става с обекта (реалният масив от елементи)? Копира ли се и той или не?

За да бъде по-нагледно обяснението, нека използваме следния пример: имаме метод **`ModifyArray()`**, който модифицира първия елемент на подаден му като параметър масив, като го реинициализира със стойност **5** и след това отпечатва елементите на масива, оградени в квадратни скоби и разделени със запетайки:

```
static void ModifyArray(int[] arrParam)
{
   arrParam[0] = 5;

   Console.Write("In ModifyArray() the param is: ");
   PrintArray(arrParam);
}

static void PrintArray(int[] arrParam)
{
   Console.Write("[");
   int length = arrParam.Length;
   if (length > 0)
   {
      Console.Write(arrParam[0].ToString());
      for (int i = 1; i < length; i++)
      {
         Console.Write(", {0}", arrParam[i]);
      }
   }
   Console.WriteLine("]");
}
```

Съответно, декларираме и метод **`Main()`**, от който извикваме новосъздадения метод **`ModifyArray()`**:

```
static void Main()
{
   int[] arrArg = new int[] { 1, 2, 3 };

```

```
    Console.Write("Before ModifyArray() the argument is: ");
    PrintArray(arrArg);

    // Modifying the array's argument
    ModifyArray(arrArg);

    Console.Write("After ModifyArray() the argument is: ");
    PrintArray(arrArg);
}
```

Какъв ще е резултатът от изпълнението на този код? Нека погледнем:

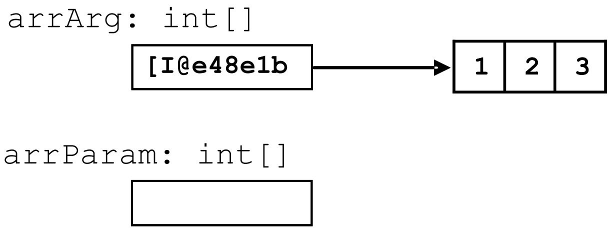
```
Before ModifyArray() the argument is: [1, 2, 3]
In ModifyArray() the param is: [5, 2, 3]
After ModifyArray() the argument is: [5, 2, 3]
```

Забелязваме, че след изпълнението на метода **`ModifyArray()`**, масивът към който променливата **`arrArg`** пази референция, не съдържа **`[1,2,3]`**, а съдържа **`[5,2,3]`**. Какво означава това?

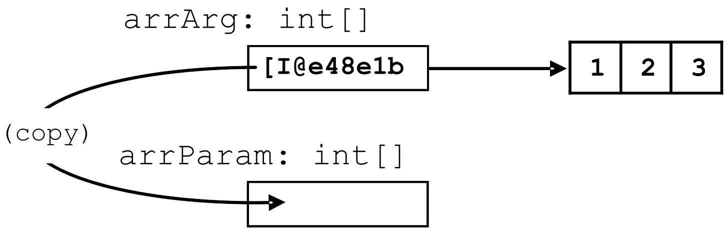
Причината за този резултат е, че при подаването на аргумент от референтен тип, се копира единствено стойността на променливата, която пази референция към обекта, но **не се прави копие на самия обект**.

При подаване на аргументи от референтен тип се копира само стойността на променливата, която пази референция към обекта в паметта, но не и самият обект.

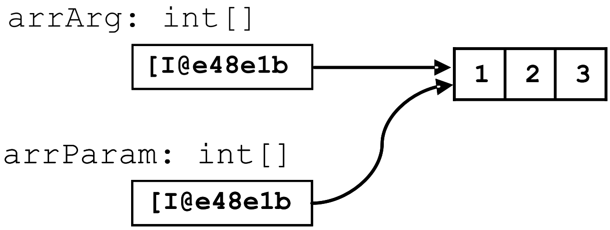
Нека онагледим казаното с няколко схеми, разглеждайки отново нашия пример. Преди извикването на метода **`ModifyArray()`**, стойността на параметъра **`arrParam`** е неопределена и той не пази референция към никакъв конкретен обект (никакъв реален масив):



По време на извикването на **`ModifyArray()`**, стойността, която е запазена в аргумента **`arrArg`**, се копира в параметъра **`arrParam`**:



По този начин, копирайки референцията към елементите на масива в паметта от аргумента в параметъра, ние указваме на параметъра да "сочи" към същия обект, към който "сочи" и аргументът:



И тъкмо това е моментът, за който трябва да сме внимателни, защото, ако извиканият метод модифицира **обекта**, към който му е подадена референция, това може да повлияе на изпълнението на кода, който следва след изпълнението на метода (както видяхме в нашия пример – методът **`PrintArray()`** не отпечата масива, който му подадохме първоначално).

Разликата между подаването на аргументи от примитивен и референтен тип се състои в начина на предаването им: **примитивните типове се предават по стойност**, а **обектите се предават по референция**.

Подаване на изрази като аргументи на метод

Когато извикваме метод, можем да подаваме цели изрази, като аргументи. Когато правим това, C# пресмята стойностите на тези изрази и по време на изпълнение (а когато е възможно още по време на компилация) заменя самия израз с пресметнатия резултат при извикването на метода. Например следният код показва извикване на методи като им подава като аргументи изрази:

```
PrintSign(2 + 3);

float oldQuantity = 3;
float quantity = 2;
PrintMax(oldQuantity * 5, quantity * 2);
```

Съответно резултатът от изпълнението на тези методи е:

```
Positive
Maximal number: 15.0
```

Когато извикваме метод с параметри, трябва да спазваме някои определени правила, които ще обясним в следващите няколко подсекции.

Подаване на аргументи съвместими с типа на съответния параметър

Трябва да знаем, че можем да подаваме аргументи, които са съвместими по тип с типа, с който е деклариран съответния параметър в списъка от параметри на метода.

Например, ако параметърът, който методът очаква в декларацията си, е от тип **`float`**, при извикването на метода, може да подадем стойност, която е от тип **`int`**. Тя ще бъде преобразувана от компилатора до стойност от тип **`float`** и едва тогава ще бъде подадена на метода и той ще бъде изпълнен:

```
static void PrintNumber(float number)
{
  Console.WriteLine("The float number is: {0}", number);
}

static void Main()
{
  PrintNumber(5);
}
```

В примера, при извикването на метода **`PrintNumber()`** в метода **`Main()`**, първо целочисленият литерал **`5`** (който по подразбиране е от тип **`int`**) се преобразува до съответната стойност с десетична запетая **`5.0f`**. След това така преобразуваната стойност се подава на метода **`PrintNumber()`**.

Както предполагаме, изходът от изпълнението на този код е:

```
The float number is: 5.0
```

Съвместимост на стойността от израз и параметър на метод

Резултатът от пресмятането на някакъв израз, подаден като аргумент, трябва да е от същия тип, какъвто е типът на параметъра в декларацията на метода или от съвместим с него тип (вж. горната точка).

Например, ако се изисква параметър от тип **`float`**, е позволено стойността от пресмятането на израза да е например от тип **`int`**. Т.е. в горния пример, ако вместо **`PrintNumber(5)`**, извикаме метода, като на мястото на **`5`**, поставим например израза **`2+3`**, резултатът от пресмятането на този израз, трябва да е от тип **`float`** (който метода очаква), или тип, който може да се преобразува до **`float`** безпроблемно (в нашия случай това е **`int`**). За да онагледим това, нека леко модифицираме метода **`Main()`** от предходната точка:

```
static void Main()
{
   PrintNumber(2 + 3);
}
```

В този пример съответно, първо ще бъде извършено сумирането, след това целочисленият резултат **5**, ще бъде преобразуван до еквивалента му с плаваща запетая **5.0f** и едва след това ще бъде извикан методът **PrintNumber(…)** с аргумент **5.0f**. Резултатът отново ще бъде:

```
The float number is: 5.0
```

Спазване на последователността на типовете на аргументите

Стойностите, които се подават на метода при неговото извикване, трябва като типове, да са в същата последователност, в каквато са параметрите на метода при неговата декларация. Това е свързано със спецификацията (сигнатурата) на метода, която дискутирахме по-горе.

За да стане по-ясно, нека разгледаме следния пример: нека имаме метод **PrintNameAndAge()**, който в декларацията си има списък от параметри, които са съответно от тип **string** и **int**, точно в тази последователност:

Person.cs

```
class Person
{
   static void PrintNameAndAge(string name, int age)
   {
      Console.WriteLine("I am {0}, {1} year(s) old.",
         name, age);
   }
}
```

Нека към нашия клас добавим метод **Main()**, в който да извикаме нашия метод **PrintNameAndAge()**, като се опитаме да му подадем аргументи, които вместо "**Pesho**" и **25**, са в обратна последователност като типове – **25** и "**Pesho**:

```
static void Main()
{
   // Wrong sequence of arguments
   Person.PrintNameAndAge(25, "Pesho");
}
```

Компилаторът няма да намери метод, който се казва **PrintNameAndAge** и в същото време приема параметри, които са последователно от тип **int** и **string**. Затова, той ще ни уведоми за грешка:

```
The best overloaded method match for
'Person.PrintNameAndAge(string, int)' has some invalid arguments
```

Метод с променлив брой аргументи (var-args)

До момента, разглеждахме деклариране на методи, при които списъкът от параметри в декларацията на метода съвпада с броя на аргументите, които му подаваме, когато го извикваме.

Сега ще разгледаме как се декларират методи, които позволяват по време на извикване, броят на подаваните аргументи да е различен, в зависимост от нуждите на извикващия код. Такива методи се наричат **методи с променлив брой аргументи**.

Нека вземем примера, който разгледахме по-горе, за пресмятане на сумата на даден масив от цени на книги. В него, като параметър на метода подавахме масив от тип **`decimal`**, в който се съхраняват цените на избраните от нас книги:

```
static void PrintTotalAmountForBooks(decimal[] prices)
{
   decimal totalAmount = 0;

   foreach (decimal singleBookPrice in prices)
   {
      totalAmount += singleBookPrice;
   }
   Console.WriteLine("The total amount of all books is:" +
      totalAmount);
}
```

Така дефиниран, този метод предполага, че винаги преди да го извикаме, ще създадем масив с числа от тип **`decimal`** и ще го инициализираме с някакви стойности.

След създаването на C# метод, който приема променлив брой параметри, е възможно, когато трябва да подадем някакъв списък от стойности от **един и същ тип** на даден метод, вместо да подаваме масив, който съдържа тези стойности, да ги подадем директно на метода като аргументи, разделени със запетая.

Например, в нашия случай с книгите, вместо да създаваме масив, специално заради извикването на този метод:

```
decimal[] prices = new decimal[] { 3m, 2.5m };
PrintTotalAmountForBooks(prices);
```

Можем директно да подадем списъка с цените на книгите, като аргументи на метода:

```
PrintTotalAmountForBooks(3m, 2.5m);
PrintTotalAmountForBooks(3m, 5.1m, 10m, 4.5m);
```

Този тип извикване на метод е възможен само ако сме декларирали метода си, така че да приема променлив брой аргументи (var-args).

Деклариране на метод с променлив брой аргументи

Формално декларацията на метод с променлив брой аргументи е същата, каквато е декларацията на всеки друг метод:

```
static <return_type> <method_name>(<parameters_list>)
{
   // Method's body
}
```

Разликата е, че **`<parameters_list>`** се декларира с ключовата дума **`params`** по следния начин:

```
<parameters_list> =
   [<type1> <name1>[, <typei> <namei>], params <var_type>[]
<var_name>]
където i= 2, 3, ...
```

Последният елемент от декларацията на списъка – **`<var_name>`**, е този, който позволява подаването на произволен брой аргументи от типа **`<var_type>`**, при всяко извикване на метода.

При декларацията на този елемент, преди типа му **`<var_type>`** трябва да добавим **`params`**: "**`params <var_type>[]`**". Типът **`<var_type>`** може да бъде както примитивен тип, така и референтен.

Правилата и особеностите за останалите елементи от списъка с параметри на метода, предхождащи var-args параметъра **`<var_name>`**, са същите, каквито ги разгледахме по-горе в тази глава.

За да стане по-ясно обясненото до момента, нека разгледаме един пример за декларация и извикване на метод с променлив брой аргументи:

```
static long CalcSum(params int[] elements)
{
   long sum = 0;
   foreach (int element in elements)
   {
      sum += element;
   }
   return sum;
}

static void Main()
```

```
{
   long sum = CalcSum(2, 5);
   Console.WriteLine(sum);

   long sum2 = CalcSum(4, 0, -2, 12);
   Console.WriteLine(sum2);

   long sum3 = CalcSum();
   Console.WriteLine(sum3);
}
```

Примерът сумира числа, като техният брой не е предварително известен. Методът може да бъде извикан с един, два или повече параметъра, а също и без параметри. Ако изпълним примера, ще получим следния резултат:

```
7
14
0
```

Същност на декларацията на параметър за променлив брой аргументи

Параметърът от формалната дефиниция по-горе, който позволява подаването на променлив брой аргументи при извикването на метода – **`<var_name>`**, всъщност е име на **масив** от тип **`<var_type>`**. При извикването на метода, аргументите от тип **`<var_type>`** или тип съвместим с него, които подаваме на метода (независимо от броя им), ще бъдат съхранени в този масив. След това те ще бъдат използвани в тялото на метода. Достъпът и работата до тези елементи става по същия начин, по който работим с масиви.

За да стане по-ясно, нека преработим метода, който пресмята сумата от цените на избраните от нас книги, да приема произволен брой аргументи:

```
static void PrintTotalAmountForBooks(params decimal[] prices)
{
   decimal totalAmount = 0;

   foreach (decimal singleBookPrice in prices)
   {
      totalAmount += singleBookPrice;
   }
   Console.WriteLine("The total amount of all books is:" +
      totalAmount);
}
```

Виждаме, че единствената промяна бе да сменим декларацията на масива **`prices`** като добавим **`params`** пред **`decimal[]`**. Въпреки това, в тялото на нашия метод, **`prices`** отново е масив от тип **`decimal`**, който използваме по познатия ни начин в тялото на метода.

Сега можем да извикаме нашия метод, без да декларираме предварително масив от числа, който да му подаваме като аргумент:

```
static void Main()
{
  PrintTotalAmountForBooks(3m, 2.5m);
  PrintTotalAmountForBooks(1m, 2m, 3.5m, 7.5m);
}
```

Съответно резултатът от двете извиквания на метода ще бъде:

```
The total amount of all books is: 5.5
The total amount of all books is: 14.0
```

Както вече се досещаме, тъй като сам по себе си **`prices`** е масив, можем да декларираме и инициализираме масив преди извикването на нашия метод и да подадем този масив като аргумент:

```
static void Main()
{
  decimal[] pricesArr = new decimal[] { 3m, 2.5m };

  // Passing initialized array as var-arg:
  PrintTotalAmountForBooks(pricesArr);
}
```

Това е напълно легално извикване и резултатът от изпълнението на този код ще е следният:

```
The total amount of all books is: 5.5
```

Позиция на декларацията на параметъра за променлив брой аргументи

Един метод, който може да приема променлив брой аргументи, **може да има и други параметри** в списъка си от параметри.

Например, следният метод, приема като първи параметър елемент от тип **`string`**, а след него нула или повече елементи от тип **`int`**:

```
static void DoSomething(string strParam, params int[] x)
{
}
```

Особеното, на което трябва да обърнем внимание е, че елементът от списъка от параметри в дефиницията на метода, който позволява подаването на произволен брой аргументи, независимо от броя на останалите параметри, трябва да е **винаги** на последно място.

Елементът от списъка от параметри на един метод, който позволява подаването на произволен брой аргументи при извикването на метода, трябва да се декларира винаги на последно място в списъка от параметри на метода.

Ако се опитаме да поставим декларацията на var-args параметъра **x**, от последния пример, да не бъде на последно място в списъка от параметри на метода:

```
static void DoSomething(params int[] x, string strParam)
{
}
```

Компилаторът ще изведе следното съобщение за грешка:

```
A parameter array must be the last parameter in a formal
parameter list
```

Ограничение на броя на параметрите за променлив брой аргументи

Друго ограничение е при методите с променлив брой аргументи, е че в декларацията на един метод не може да имаме повече от един параметър, който позволява подаването на променлив брой аргументи. Така, ако се опитаме да компилираме следната декларация на метод:

```
static void DoSomething(params int[] x, params string[] z)
{
}
```

Компилаторът ще изведе отново познатото съобщение за грешка:

```
A parameter array must be the last parameter in a formal
parameter list
```

Това правило е частен случай на правилото за позицията на var-args параметъра – да бъде на последно място в списъка от параметри.

Особеност при извикване на метод с променлив брой параметри, без подаване на нито един параметър

След като се запознахме с декларацията и извикването на методи с променлив брой аргументи и разбрахме същността им, може би възниква въпроса, какво ще стане, ако не подадем нито един аргумент на такъв метод по време на извикването му?

Например, какъв ще е резултатът от изпълнението на нашия метод за пресмятане цената на избраните от нас книги, в случая, когато не сме си харесали нито една книга:

```
static void Main()
{
  PrintTotalAmountForBooks();
}
```

Виждаме, че компилацията на този код минава без проблеми и след изпълнението резултатът е следният:

```
The total amount of all books is: 0
```

Това е така, защото, въпреки че не сме подали нито една стойност на нашия метод, при извикването на метода, масивът **`decimal[] prices`** е създаден, но е празен (т.е. не съдържа нито един елемент).

Това е добре да бъде запомнено, тъй като дори да няма подадени стойности, C# се грижи да **инициализира масива**, в който се съхраняват променливия брой аргументи.

Метод променлив брой параметри – пример

Имайки предвид как дефинираме методи с променлив брой аргументи, можем да запишем добре познатият ни **`Main()`** метод по следния начин:

```
public static void Main(params String[] args)
{
  // Method body comes here
}
```

Горната дефиниция е напълно валидна и се приема без проблеми от компилатора.

Именувани и незадължителни параметри

Именуваните и незадължителните параметри са две отделни възможности на езика, но често се използват заедно. Те са нововъведение в C# версия 4.0. **Незадължителните параметри** позволяват пропускането на параметри при извикване на метод. **Именуваните параметри** позволяват да бъде подадена стойност на параметър чрез името му вместо да се разчита на позицията му в списъка от параметрите. Тези нови възможности в синтаксиса на езика C# са особено полезни когато искаме да позволим даден метод да бъде извикван с различни комбинации от параметри.

Декларирането на незадължителен параметър става просто чрез осигуряване на стойност по подразбиране за него по следния начин:

```
static void SomeMethod(int x, int y = 5, int z = 7)
```

```
{
}
```

В горния пример **`y`** и **`z`** са незадължителни параметри и могат да бъдат пропуснати при извикване на метода:

```
static void Main()
{
   // Normal call of SomeMethod
   SomeMethod(1, 2, 3);
   // Omitting z - equivalent to SomeMethod(1, 2, 7)
   SomeMethod(1, 2);
   // Omitting both y and z - equivalent to SomeMethod(1, 5, 7)
   SomeMethod(1);
}
```

Подаването на стойности на параметри по име става чрез задаване на името на параметъра, следвано от двоеточие и от стойността на параметъра. Ето един пример:

```
static void Main()
{
   // Passing z by name
   SomeMethod(1, z: 3);
   // Passing both x and z by name
   SomeMethod(x: 1, z: 3);
   // Reversing the order of the arguments
   SomeMethod(z: 3, x: 1);
}
```

Всички извиквания в горния пример са еквивалентни – пропуска се параметърът **`y`**, а като стойности на параметрите **`x`** и **`z`** се подават съответно 1 и 3. Единствената разлика е, че стойностите на параметрите се изчисляват в реда в който са подадени, така че в последното извикване 3 се изчислява преди 1 (в случая 3 е просто константа, но ако е някакъв по-сложен израз, редът на изчисление би могъл да е от значение).

Варианти на методи (method overloading)

Когато в даден клас декларираме един метод, чието име съвпада с името на друг метод, но сигнатурите на двата метода се различават по **списъка от параметри** (броят на елементите в него или подредбата им), казваме, че имаме различни **варианти на този метод** (**method overloading**).

Например, да си представим, че имаме задачата да напишем програма, която рисува на екрана букви и цифри. Съответно можем да си представим, че нашата програма, може да има методите за рисуване съответно на низове **`DrawString(string str)`**, на цели числа – **`DrawInt(int number)`**, на десетични числа – **`DrawFloat(float number)`** и т.н.:

```
static void DrawString(string str)
{
   // Draw string
}

static void DrawInt(int number)
{
   // Draw integer
}

static void DrawFloat(float number)
{
   // Draw float number
}
```

Но езикът C# позволява да си създадем съответно само варианти на един и същ метод **`Draw(…)`**, който приема комбинации от различни типове параметри, в зависимост от това, какво искаме да нарисуваме на екрана:

```
static void Draw(string str)
{
   // Draw string
}

static void Draw(int number)
{
   // Draw integer
}

static void Draw(float number)
{
   // Draw float number
}
```

Горната дефиниция на методи е валидна и се компилира без грешки. Методът **`Draw(…)`** от примера се нарича **предефиниран** (**overloaded**).

Значение на параметрите в сигнатурата на метода

Както обяснихме по-горе, за спецификацията (сигнатурата) на един метод, в C#, единствените елементи от списъка с параметри, които имат значение, са **типовете на параметрите** и **последователността, в която са изброени**. Имената на параметрите нямат значение за еднозначното деклариране на метода.

За еднозначното деклариране на метод в C#, по отношение на списъка с параметри на метода, единствено има значение неговата сигнатура, т.е.:

- **типът на параметрите на метода**

- **последователността на типовете в списъка от параметри**

Имената на параметрите не се вземат под внимание.

Например за C#, следните две декларации, са декларации на един и същ метод, тъй като типовете на параметрите в списъка от параметри са едни и същи – **`int`** и **`float`**, независимо от имената на променливите, които сме поставили – **`param1`** и **`param2`** или **`arg1`** и **`arg2`**:

```
static void DoSomething(int param1, float param2) { }
static void DoSomething(int arg1, float arg2) { }
```

Ако декларираме два метода в един и същ клас, по този начин, компилаторът ще изведе съобщение за грешка, подобно на следното:

```
Type '<the_name_of_your_class>' already defines a member called
'DoSomething' with the same parameter types.
```

Ако обаче в примера, който разгледахме, някои от **параметрите на една и съща позиция в списъка от параметри са от различен тип**, тогава за C#, това са два напълно различни метода, или по-точно, напълно различни **варианти на метод с даденото име**.

Например, ако във втория метод, вторият параметър от списъка на единия от методите – **`float arg2`**, го декларираме да не бъде от тип **`float`**, а **`int`**, тогава това ще бъдат два различни метода с различна сигнатура – **`DoSomething(int, float)`** и **`DoSomething(int, int)`**. Вторият елемент от сигнатурата им – **списъкът от параметри**, е напълно различен, тъй като типовете на вторите им елементи от списъка са различни:

```
static void DoSomething(int arg1, float arg2) { }
static void DoSomething(int param1, int param2) { }
```

В този случай, дори да поставим едни и същи имена на параметрите в списъка, компилаторът ще ги приеме, тъй като за него това са различни методи:

```
static void DoSomething(int param1, float param2) { }
static void DoSomething(int param1, int param2) { }
```

Компилаторът отново "няма възражения", ако декларираме вариант на метод, но този път вместо да подменяме типа на втория параметър, просто разменим местата на параметрите на втория метод:

```
static void DoSomething(int param1, float param2) { }
static void DoSomething(float param2, int param1) { }
```

Тъй като **последователността на типовете** на параметрите в списъка с параметри е различна, съответно и спецификациите (сигнатурите) на методите са различни. Щом списъците с параметри са различни, то еднаквите имена (**DoSomething**) нямат отношение към еднозначното деклариране на методите в нашия клас – имаме различни сигнатури.

Извикване на варианти на методи (overloaded methods)

След като веднъж сме декларирали методи със съвпадащи имена и различна сигнатура, след това можем да ги извикваме като всички други методи – чрез име и подавани аргументи. Ето един пример:

```
static void PrintNumbers(int intValue, float floatValue)
{
  Console.WriteLine(intValue + "; " + floatValue);
}

static void PrintNumbers(float floatValue, int intValue)
{
  Console.WriteLine(floatValue + "; " + intValue);
}

static void Main()
{
  PrintNumbers(2.71f, 2);
  PrintNumbers(5, 3.14159f);
}
```

Ако изпълним кода от примера, ще се убедим, че при първото извикване се извиква втория метод, а при второто извикване се извиква първия метод. Кой метод да се извика зависи от типа на подадените параметри. Резултатът от изпълнението на горния код е следният:

```
2.71; 2
5; 3.14159
```

Ако се опитаме, обаче да направим следното извикване, ще получим грешка:

```
static void Main()
{
  PrintNumbers(2, 3);
}
```

Причината за грешката е, че компилаторът се опитва да преобразува двете цели числа към подходящи типове, за да ги подаде на един от двата метода с име **PrintNumbers**, но съответните преобразувания не са еднозначни. Има два варианта – или първият параметър да се преобразува към **float** и да се извика методът **PrintNumbers(float, int)** или вторият параметър да се преобразува към **float** и да се извика методът

`PrintNumbers(int, float)`. Това е нееднозначност, която компилаторът изисква да бъде разрешена ръчно, например по следния начин:

```
static void Main()
{
  PrintNumbers((float)2, (short)3);
}
```

Горния код ще се компилира успешно, тъй като след преобразованието на аргументите, става еднозначно кой точно метод да бъде извикан – **`PrintNumbers(float, int)`**.

Методи със съвпадащи сигнатури

Накрая, преди да продължим със няколко интересни примера за използване на методи, нека да разгледаме следния пример за некоректно предефиниране (overload) на методи:

```
static int Sum(int a, int b)
{
    return a + b;
}

static long Sum(int a, int b)
{
    return a + b;
}

static void Main()
{
    Console.WriteLine(Sum(2, 3));
}
```

Кодът от примера ще предизвика грешка при компилация, тъй като имаме два метода с еднакви списъци от параметри (т.е. с еднаква сигнатура), които обаче връщат различен тип резултат. При опит за извикване се получава нееднозначие, което не може да бъде разрешено от компилатора.

Триъгълници с различен размер – пример

След като разгледахме как да декларираме и извикваме методи с параметри и как да връщане резултати от извикване на метод, нека сега дадем един по-цялостен пример, с който да покажем къде може да се използват методите с параметри. Да предположим, че искаме да напишем програма, която отпечатва триъгълници, като тези, показани по-долу:

```
                                       1
          1                            1 2
          1 2                          1 2 3
          1 2 3                        1 2 3 4
          1 2 3 4                 1 2 3 4 5
n=5  ->   1 2 3 4 5     n=6  ->   1 2 3 4 5 6
          1 2 3 4                 1 2 3 4 5
          1 2 3                   1 2 3 4
          1 2                     1 2 3
          1                       1 2
                                  1
```

Едно възможно решение на задачата е дадено по-долу:

Triangle.cs

```
using System;

class Triangle
{
  static void Main()
  {
    // Entering the value of the variable n
    Console.Write("n = ");
    int n = int.Parse(Console.ReadLine());
    Console.WriteLine();

    // Printing the upper part of the triangle
    for (int line = 1; line <= n; line++)
    {
      PrintLine(1, line);
    }

    // Printing the bottom part of the triangle
    // that is under the longest line
    for (int line = n - 1; line >= 1; line--)
    {
      PrintLine(1, line);
    }
  }

  static void PrintLine(int start, int end)
  {
    for (int i = start; i <= end; i++)
    {
      Console.Write(" " + i);
    }
    Console.WriteLine();
  }
}
```

Нека разгледаме как работи примерното решение. Тъй като, можем да печатаме в конзолата ред по ред, разглеждаме триъгълниците, като поредици числа, разположени в отделни редове. Следователно, за да ги изведем в конзолата, трябва да имаме средство, което извежда отделните редове от триъгълниците. За целта, създаваме метода **`PrintLine(…)`**.

В него, с помощта на цикъл **`for`**, отпечатваме в конзолата редица от последователни числа. Първото число от тази редица е съответно първият параметър от списъка с параметри на метода (променливата **`start`**). Последният елемент на редицата е числото, подадено на метода, като втори параметър (променливата **`end`**).

Забелязваме, че тъй като числата са последователни, дължината (броят числа) на всеки ред, съответства на разликата между втория параметър **`end`** и първия – **`start`**, от списъка с параметри на метода (това ще ни послужи малко по-късно, когато конструираме триъгълниците).

След това създаваме алгоритъм за отпечатването на триъгълниците, като цялостни фигури, в метода **`Main()`**. Чрез метода **`int.Parse`** въвеждаме стойността на променливата **`n`** и извеждаме празен ред.

След това, в два последователни **`for`**-цикъла конструираме триъгълника, който трябва да се изведе, за даденото **`n`**. В първия цикъл отпечатваме последователно всички редове от горната част на триъгълника до средния – най-дълъг ред, включително. Във втория цикъл, отпечатваме редовете на триъгълника, които трябва да се изведат под средния (най-дълъг) ред.

Както отбелязахме по-горе, номерът на реда, съответства на броя на елементите (числа) намиращи се на съответния ред. И тъй като винаги започваме от числото **1**, номерът на реда, в горната част от триъгълника, винаги ще е равен на последния елемент на редицата, която трябва да се отпечата на дадения ред. Следователно, можем да използваме това при извикването на метода **`PrintLine(…)`**, тъй като той изисква точно тези параметри за изпълнението на задачата си.

Прави ни впечатление, че броят на елементите на редиците, се увеличава с единица и съответно, последният елемент на всяка по-долна редица, трябва да е с единица по-голям от последния елемент на редицата от предходния ред. Затова, при всяко "завъртане" на първия for-цикъл, подаваме на метода **`PrintLine(…)`**, като първи параметър **1**, а като втори – текущата стойност на променливата **`line`**. Тъй като при всяко изпълнение на тялото на цикъла **`line`** се увеличава с единица, на при всяка итерация методът **`PrintLine(…)`** ще отпечатва редица с един елемент повече от предходния ред.

При втория цикъл, който отпечатва долната част на триъгълника, следваме обратната логика. Колкото по-надолу печатаме, редиците трябва да се смаляват с по един елемент и съответно последният елемент на всяка редица, трябва да е с единица по-малък от последния елемент на редицата от предходния ред. От тук задаваме началното условие за стойността на променливата **`line`** във втория цикъл: **`line = n-1`**. След всяко завъртане на цикъла намаляваме стойността на **`line`** с единица и я подаваме като втори параметър на **`PrintLine(…)`**.

Още едно подобрение, което можем да направим, е да изнесем логиката, която отпечатва един триъгълник в отделен метод. Забелязваме, че логически, печатането на триъгълник е ясно обособено, затова можем да декларираме метод с един параметър (стойността, която въвеждаме от клавиатурата) и да го извикаме в метода **`Main()`**:

```
static void Main()
{
   Console.Write("n = ");
   int n = int.Parse(Console.ReadLine());
   Console.WriteLine();

   PrintTriangle(n);
}

static void PrintTriangle(int n)
{
   // Printing the upper part of the triangle
   for (int line = 1; line <= n; line++)
   {
      PrintLine(1, line);
   }

   // Printing the bottom part of the triangle
   // that is under the longest line
   for (int line = n - 1; line >= 1; line--)
   {
      PrintLine(1, line);
   }
}
```

Ако изпълним програмата и въведем за **`n`** стойност 3, ще получим следния резултат:

```
n = 3

 1
 1 2
 1 2 3
 1 2
 1
```

Връщане на резултат от метод

До момента, винаги давахме примери, в които методът извършва някакво действие, евентуално отпечатва нещо в конзолата, приключва работата си и с това се изчерпват "задълженията" му. Един метод, обаче, освен просто да изпълнява списък от действия, може да върне някакъв резултат от изпълнението си. Нека разгледаме как става това.

Деклариране на метод с връщана стойност

Ако погледнем отново как декларираме метод:

```
static <return_type> <method_name>(<parameters_list>)
```

Ще си припомним, че когато обяснявахме за това, споменахме, че на мястото на **`<return type>`** поставяме **`void`**. Сега ще разширим дефиницията, като кажем, че на това място може да стои не само **`void`**, но и произволен тип – примитивен (**`int`**, **`float`**, **`double`**, …) или референтен (например **`string`** или масив), в зависимост от какъв тип е резултатът от изпълнението на метода.

Например, ако вземем примера с метода, който изчислява лице на квадрат, вместо да отпечатваме стойността в конзолата, методът може да я върне като резултат. Ето как би изглеждала декларацията на метода:

```
static double CalcSquareSurface(double sideLength)
```

Вижда се, че резултатът от пресмятането на лицето е от тип **`double`**.

Употреба на връщаната стойност

Когато методът бъде изпълнен и върне стойност, можем да си представяме, че C# поставя тази стойност на мястото, където е било извикването на метода и продължава работа с нея. Съответно, тази върната стойност, можем да използваме от извикващия метод за най-различни цели.

Присвояване на променлива

Може да присвоим резултата от изпълнението на метода на променлива от подходящ тип:

```
// GetCompanyLogo() returns a string
string companyLogo = GetCompanyLogo();
```

Употреба в изрази

След като един метод върне резултат, този резултат може да бъде използван в изрази.

Например, за да намерим общата цена при пресмятане на фактури, трябва да получим единичната цена и да умножим по количеството:

```
float totalPrice = GetSinglePrice() * quantity;
```

Подаване като стойност в списък от параметри на друг метод

Можем да подадем резултата от работата на един метод като стойност в списъка от параметри на друг метод:

```
Console.WriteLine(GetCompanyLogo());
```

В този пример, отначало извикваме метода **`GetCompanyLogo()`**, подавайки го като аргумент на метода **`WriteLine()`**. Веднага, след като методът **`GetCompanyLogo()`** бъде изпълнен, той ще върне резултат, например "**`Microsoft Corporation`**". Тогава C# ще "подмени" извикването на метода, с резултата, който е върнат от изпълнението му и можем да приемем, че в кода имаме:

```
Console.WriteLine("Microsoft Corporation");
```

Тип на връщаната стойност

Както обяснихме малко по-рано, резултатът, който връща един метод, може да е от всякакъв тип – **`int`**, **`string`**, масив и т.н. Когато обаче, като тип на връщаната стойност бъде употребена ключовата дума **`void`**, с това означаваме, че методът не връща никаква стойност.

Операторът return

За да накараме един метод да връща стойност, трябва в тялото му, да използваме ключовата дума **`return`**, следвана от израз, който да бъде върнат като резултат от метода:

```
static <return_type> <method_name>(<parameters_list>)
{
  // Some code that is preparing the method’s result comes here
  return <method’s_result>;
}
```

Съответно **`<method’s_result>`**, е от тип **`<return_type>`**. Например:

```
static long Multiply(int number1, int number2)
{
   long result = number1 * number2;
   return result;
}
```

В този метод, след умножението, благодарение на **`return`**, методът ще върне като резултат от изпълнението на метода целочислената променлива **`result`**.

Резултат от тип, съвместим, с типа на връщаната стойност

Резултатът, който се връща от метода, може да е от тип, който е **съвместим** (който може неявно да се преобразува) с типа на връщаната стойност **`<return_type>`**.

Например, може да модифицираме последния пример, в който типа на връщаната стойност да е от тип **`float`**, а не **`int`** и да запазим останалия код по следния начин:

```
static float Multiply(int number1, int number2)
{
   int result = number1 * number2;
   return result;
}
```

В този случай, след изпълнението на умножението, резултатът ще е от тип **`int`**. Въпреки това, на реда, на който връщаме стойността, той ще бъде неявно преобразуван до дробно число от тип **`float`** и едва тогава, ще бъде върнат като резултат.

Поставяне на израз след оператора return

Позволено е (когато това няма да направи кода трудно четим) след ключовата дума **`return`**, да поставяме директно изрази:

```
static int Multiply(int number1, int number2)
{
   return number1 * number2;
}
```

В тази ситуация, след като изразът **`number1 * number2`** бъде изчислен, резултатът от него ще бъде заместен на мястото на израза и ще бъде върнат от оператора **`return`**.

Характеристики на оператора return

При изпълнението си операторът **`return`** извършва две неща:

- **Прекратява** изпълнението на метода.
- Връща резултата от изпълнението на метода към извикващия метод.

Във връзка с първата характеристика на оператора **`return`**, трябва да отбележим, че тъй като той прекратява изпълнението на метода, след него, до затварящата скоба, не трябва да има други оператори.

Ако все пак направим това, компилаторът ще покаже предупреждение:

```
static int Add(int number1, int number2)
{
   int result = number1 + number2;
   return result;

   // Let us try to "clean" the result variable here:
   result = 0;
}
```

В този пример компилацията ще е успешна, но за редовете след **`return`**, компилаторът ще изведе предупреждение, подобно на следното:

```
Unreachable code detected
```

Когато методът има тип на връщана стойност **`void`**, тогава след **`return`**, не трябва да има израз, който да бъде върнат. В този случай употребата на **`return`** е единствено за излизане от метода:

```
static void PrintPositiveNumber(int number)
{
   if (number <= 0)
   {
      // If the number is NOT positive, terminate the method
      return;
   }
   Console.WriteLine(number);
}
```

Последното, което трябва да научим за оператора **`return`** е, че може да бъде извикван от няколко места в метода, като е гарантирано, че всеки следващ оператор **`return`** е достъпен при определени входни условия.

Нека разгледаме примера за метод, който получава като параметри две числа и в зависимост дали първото е по-голямо от второто, двете са равни, или второто е по-голямо от първото, връща съответно 1, 0 и -1:

```
static int CompareTo(int number1, int number2)
{
   if (number1 > number2)
   {
      return 1;
   }
   else if (number1 == number2)
   {
      return 0;
   }
   else
   {
      return -1;
   }
}
```

Защо типът на връщаната стойност не е част от сигнатурата на метода?

В C# не е позволено да имаме няколко метода, които имат еднакви име и параметри, но различен тип на връщаната стойност. Това означава, че следния код няма да се компилира:

```
static int Add(int number1, int number2)
{
  return (number1 + number2);
}

static double Add(int number1, int number2)
{
  return (number1 + number2);
}
```

Причината за това ограничение е, че компилаторът не знае кой от двата метода да извика, когато се наложи, и няма как да разбере. Затова, още при опита за декларация на двата метода, той ще изведе следното съобщение за грешка:

```
Type '<the_name_of_your_class>' already defines a member called
'Add' with the same parameter types
```

където **`<the_name_of_your_class>`** е името на класа, в който се опитваме да декларираме двата метода.

Преминаване от Фаренхайт към Целзий – пример

В следващата задача се изисква да напишем програма, която при подадена от потребителя телесна температура, измерена в градуси по Фаренхайт, да я преобразува и изведе в съответстващата ѝ температура в градуси по Целзий със следното съобщение: "**`Your body temperature in Celsius degrees is X`**", където **`X`** е съответно градусите по Целзий. В допълнение, ако измерената температура в градуси Целзий е по-висока от 37 градуса, програмата трябва да предупреждава потребителя, че е болен, със съобщението "**`You are ill!`**".

Като за начало можем да направим бързо проучване в Интернет и да прочетем, че формулата за преобразуване на температури е **`°C = (°F - 32) * 5 / 9`**, където съответно с **`°C`** отбелязваме температурата в градуси Целзий, а с **`°F`** – съответно тази в градуси Фаренхайт.

Анализираме поставената задача и виждаме, че подзадачките, на които може да се раздели са следните:

- Вземаме температурата измервана в градуси по Фаренхайт като вход от клавиатурата (потребителят ще трябва да я въведе).
- Преобразуваме полученото число в съответното му число за температурата, измервана в градуси по Целзий.
- Извеждаме съобщение за преобразуваната температура в Целзий.
- Ако температурата е по-висока от 37 ºC, извеждаме съобщение на потребителя, че той е болен.

Ето едно примерно решение:

TemperatureConverter.cs

```
using System;

class TemperatureConverter
{
  static double ConvertFahrenheitToCelsius(double temperatureF)
  {
    double temperatureC = (temperatureF - 32) * 5 / 9;
    return temperatureC;
  }

  static void Main()
  {
    Console.Write(
      "Enter your body temperature in Fahrenheit degrees: ");
    double temperature = double.Parse(Console.ReadLine());

    temperature = ConvertFahrenheitToCelsius(temperature);

    Console.WriteLine(
      "Your body temperature in Celsius degrees is {0}.",
      temperature);

    if (temperature >= 37)
    {
      Console.WriteLine("You are ill!");
    }
  }
}
```

Операциите по въвеждането на температурата и извеждането на съобщенията са тривиални, и за момента прескачаме решението им, като се съсредоточаваме върху преобразуването на температурите. Виждаме, че това е логически обособено действие, което може да отделим в метод. Това, освен че ще направи кода ни по-четим, ще ни даде възможност в бъдеще, да правим подобно преобразование отново като преизползваме този метод. Декларираме метода **ConvertFahrenheitToCelsius(…)**, със списък от един параметър с името **temperatureF**, който представлява измерената температура в градуси по Фаренхайт и връща съответно число от тип **double**, което представлява преобразуваната температура в градуси по Целзий. В тялото му ползваме намерената в Интернет формула чрез синтаксиса на C#.

След като сме приключили с тази стъпка от решението на задачата, решаваме, че останалите стъпки няма нужда да ги извеждаме в методи, а е достатъчно да ги имплементираме в метода **Main()** на класа.

С помощта на метода **`double.Parse(…)`**, получаваме телесната температура на потребителя, като предварително сме го попитали за нея със съобщението "**`Enter your body temperature in Fahrenheit degrees`**".

След това извикваме метода **`ConvertFahrenheitToCelsius()`** и съхраняваме върнатия резултат в променливата **`temperature`**.

С помощта на метода **`Console.WriteLine()`** извеждаме съобщението "**`Your body temperature in Celsius degrees is X`**", където **`X`** заменяме със стойността на **`temperature`**.

Последната стъпка, която трябва да се направи е с условната конструкция **`if`**, да проверим дали температурата е по-голяма или равна на 37 градуса Целзий и ако е, да изведем съобщението, че потребителят е болен.

Ето примерен изход от програмата:

```
Enter your body temperature in Fahrenheit degrees: 100
Your body temperature in Celsius degrees is 37,777778.
You are ill!
```

Разстояние между два месеца – пример

Да разгледаме следната задача: искаме да напишем програма, която при зададени две числа, които трябва да са между **`1`** и **`12`**, за да съответстват на номер на месец от годината, да извежда броя месеци, които делят тези два месеца. Съобщението, което програмата трябва да отпечатва в конзолата трябва да е "**`There is X months period from Y to Z.`**", където **`X`** е броят на месеците, който трябва да изчислим, а **`Y`** и **`Z`**, са съответно имената на месеците за начало и край на периода.

Прочитаме задачата внимателно и се опитваме да я разбием на подпроблеми, които да решим лесно и след това интегрирайки решенията им в едно цяло да получим решението на цялата задача. Виждаме, че трябва да решим следните подзадачки:

- Да въведем номерата на месеците за начало и край на периода.
- Да пресметнем периода между въведените месеци.
- Да изведем съобщението.
- В съобщението вместо числата, които сме въвели за начален и краен месец на периода, да изведем съответстващите им имена на месеци на английски.

Ето едно възможно решение на поставената задача:

Months.cs

```
using System;
```

```
class Months
{
   static string GetMonth(int month)
   {
      string monthName;
      switch (month)
      {
      case 1:
         monthName = "January";
         break;
      case 2:
         monthName = "February";
         break;
      case 3:
         monthName = "March";
         break;
      case 4:
         monthName = "April";
         break;
      case 5:
         monthName = "May";
         break;
      case 6:
         monthName = "June";
         break;
      case 7:
         monthName = "July";
         break;
      case 8:
         monthName = "August";
         break;
      case 9:
         monthName = "September";
         break;
      case 10:
         monthName = "October";
         break;
      case 11:
         monthName = "November";
         break;
      case 12:
         monthName = "December";
         break;
      default:
         Console.WriteLine("Invalid month!");
         return null;
      }
      return monthName;
   }
```

```
    static void SayPeriod(int startMonth, int endMonth)
    {
        int period = endMonth - startMonth;
        if (period < 0)
        {
            // Fix negative distance
            period = period + 12;
        }
        Console.WriteLine(
            "There is {0} months period from {1} to {2}.",
            period, GetMonth(startMonth), GetMonth(endMonth));
    }

    static void Main()
    {
        Console.Write("First month (1-12): ");
        int firstMonth = int.Parse(Console.ReadLine());

        Console.Write("Second month (1-12): ");
        int secondMonth = int.Parse(Console.ReadLine());

        SayPeriod(firstMonth, secondMonth);
    }
}
```

Решението на първата подзадача е тривиално. В метода **`Main()`** използваме метода **`int.Parse(…)`** и получаваме номерата на месеците за периода, чиято дължина искаме да пресметнем.

След това забелязваме, че пресмятането на периода и отпечатването на съобщението може да се обособи логически като подзадачка, и затова създаваме метод **`SayPeriod(…)`** с два параметъра – числа, съответстващи на номерата на месеците за начало и край на периода. Той няма да връща стойност, но ще пресмята периода и ще отпечатва съобщението описано в условието на задачата с помощта на стандартния изход – **`Console.WriteLine(…)`**.

Очевидното решение, за намирането на дължината на периода между два месеца, е като извадим поредния номер на началния месец от този на месеца за край на периода. Съобразяваме обаче, че ако номерът на втория месец е по-малък от този на първия, тогава потребителят е имал предвид, че вторият месец, не се намира в текущата година, а в следващата. Затова, ако разликата между двата месеца е отрицателна, към нея добавяме **`12`** – дължината на една година в брой месеци, и получаваме дължината на търсения период. След това извеждаме съобщението, като за отпечатването на имената на месеците, чийто пореден номер получаваме от потребителя, използваме метода **`GetMonth(…)`**.

Методът за извличане на име на месец по номера му можем да реализираме чрез условната конструкция **`switch-case`**, с която да съпоставим на всяко число, съответстващото му име на месец от годината. Ако стойността на входния параметър не е някоя между стойностите **1** и **12**, съобщаваме за грешка. По-нататък в главата "Обработка на изключения" ще обясним как можем да съобщаваме за грешка по-начин, който позволява грешката да бъде прихващана и обработвана, но за момента просто ще отпечатваме съобщение за грешка на конзолата.

Накрая, в метода **`Main()`** извикваме метода **`SayPeriod()`**, подавайки му въведените от потребителя числа за начало и край на периода и с това сме решили задачата.

Ето какъв би могъл да е изходът от програмата при входни данни 2 и 6:

```
First month (1-12): 2
Second month (1-12): 6
There is 4 months period from February to June.
```

Валидация на данни – пример

В тази задача, трябва да напишем програма, която пита потребителя колко е часът (с извеждане на въпроса "**`What time is it?`**"). След това потребителят, трябва да въведе две числа, съответно за час и минути. Ако въведените данни представляват валидно време, програмата, трябва да изведе съобщението "**`The time is HH:mm now.`**", където с **`HH`** съответно сме означили часа, а с **`mm`** – минутите. Ако въведените час или минути не са валидни, програмата трябва да изведе съобщението "**`Incorrect time!`**".

След като прочитаме условието на задачата внимателно, стигаме до извода, че решението на задачата може да се разбие на следните подзадачи:

- Получаване на входа за час и минути.
- Проверка на валидността на входните данни.
- Извеждаме съобщение за грешка или валидно време.

Знаем, че обработката на входа и извеждането на изхода няма да бъдат проблем за нас, затова решаваме да се фокусираме върху проблема с валидността на входните данни, т.е. валидността на числата за часове и минути. Знаем, че часовете варират от 0 до 23 включително, а минутите съответно от 0 до 59 включително. Тъй като данните (часове и минути) не са еднородни решаваме да създадем два отделни метода, единият от които проверява валидността на часовете, а другия – на минутите.

Ето едно примерно решение:

DataValidation.cs

```
using System;
```

```
class DataValidation
{
    static void Main()
    {
        Console.WriteLine("What time is it?");

        Console.Write("Hours: ");
        int hours = int.Parse(Console.ReadLine());

        Console.Write("Minutes: ");
        int minutes = int.Parse(Console.ReadLine());

        bool isValidTime =
            ValidateHours(hours) && ValidateMinutes(minutes);
        if (isValidTime)
        {
            Console.WriteLine("The time is {0}:{1} now.",
                hours, minutes);
        }
        else
        {
            Console.WriteLine("Incorrect time!");
        }
    }

    static bool ValidateHours(int hours)
    {
        bool result = (hours >= 0) && (hours < 24);
        return result;
    }

    static bool ValidateMinutes(int minutes)
    {
        bool result = (minutes >= 0) && (minutes <= 59);
        return result;
    }
}
```

Методът, който проверява часовете, именуваме **`ValidateHours()`**, като той приема едно число от тип **`int`** за часовете и връща резултат от тип **`bool`**, т.е. **`true`** ако въведеното число е валиден час и **`false`** в противен случай:

```
static bool ValidateHours(int hours)
{
    bool result = (hours >= 0) && (hours < 24);
    return result;
}
```

По подобен начин, декларираме метод, който проверява валидността на минутите. Наричаме го **`ValidateMinutes()`**, като той приема един параметър цяло число за минутите и има тип на връщана стойност – **bool**. Ако въведеното число удовлетворява условието, което описахме по-горе (да е между 0 и 59 включително), методът ще върне като резултат **true**, а иначе – **false**:

```
static bool ValidateMinutes(int minutes)
{
  bool result = (minutes >= 0) && (minutes <= 59);
  return result;
}
```

След като сме готови с най-сложната част от задачата, декларираме метода **Main()**. В тялото му, извеждаме въпроса според условието на задачата – "**What time is it?**". След това с помощта на метода **int.Parse(…)**, прочитаме от потребителя числата за часове и минути, като резултатите ги съхраняваме в целочислените променливи, съответно **hours** и **minutes**:

```
Console.WriteLine("What time is it?");

Console.Write("Hours: ");
int hours = int.Parse(Console.ReadLine());

Console.Write("Minutes: ");
int minutes = int.Parse(Console.ReadLine());
```

Съответно, резултата от валидацията съхраняваме в променлива от тип **bool** – **isValidTime**, като последователно извикваме методите, които вече декларирахме – **ValidateHours()** и **ValidateMinutes()**, като съответно им подаваме като аргументи променливите **hours** и **minutes**. За да ги валидираме едновременно, обединяваме резултатите от извикването на методите с оператора за логическо "и" **&&**:

```
bool isValidTime =
  ValidateHours(hours) && ValidateMinutes(minutes);
```

След като сме съхранили резултата, дали въведеното време е валидно или не, в променливата **isValidTime**, го използваме в условната конструкция **if**, за да изпълним и последния подпроблем от цялостната задача – извеждането на информация към потребителя дали времето, въведено от него е валидно или не. С помощта на **Console.WriteLine(…)**, ако **isValidTime** е **true**, на конзолата извеждаме "**The time is HH:mm now.**", където **HH** е съответно стойността на променливата **hours**, а **mm** – тази на променливата **minutes**. Съответно в **else** частта от условната конструкция извеждаме, че въведеното време е невалидно – "**Incorrect time!**".

Ето как изглежда изходът от програмата при въвеждане на коректни данни:

```
What time is it?
Hours: 17
Minutes: 33
The time is 17:33 now.
```

Ето какво се случва при въвеждане на некоректни данни:

```
What time is it?
Hours: 33
Minutes: -2
Incorrect time!
```

Сортиране на числа – пример

Нека се опитаме да създадем метод, който сортира (подрежда по големина) във възходящ ред подадени му числа и като резултат връща масив със сортираните числа.

При тази формулировка на задачата, се досещаме, че подзадачите, с които трябва да се справим са две:

- По какъв начин да подадем на нашия метод числата, които трябва да сортираме.
- Как да извършим сортирането на тези числа.

Това, че трябва да върнем като резултат от изпълнението на метода, масив със сортираните числа, ни подсказва, че може да декларираме метода да приема масив от числа, който масив в последствие да сортираме, а след това да върнем като резултат:

```
static int[] Sort(int[] numbers)
{
   // The sorting logic comes here...

   return numbers;
}
```

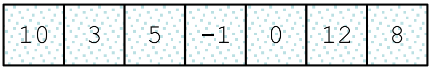
Това решение изглежда, че удовлетворява изискванията от задачата ни, но се досещаме, че може да го оптимизираме малко и вместо методът да приема като един аргумент числов масив, може да го декларираме, да приема произволен брой числови параметри.

Това ще ни спести предварителното инициализиране на масив преди извикването на метода при по-малък брой числа за сортиране, а когато числата са по-голям брой, както видяхме в секцията за деклариране ма метод с произволен брой аргументи, директно можем да подадем на метода инициализиран масив от числа, вместо да ги изброяваме като параметри на метода. Така първоначалната декларация на метода ни приема следния вид:

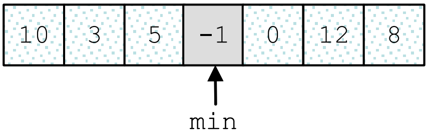
```
static int[] Sort(params int[] numbers)
{
   // The sorting logic comes here...

   return numbers;
}
```

Сега трябва да решим как да сортираме нашия масив. Един от най-лесните начини това да бъде направено е чрез така наречения **метод на пряката селекция (selection sort algorithm)**. При него масивът се разделя на сортирана и несортирана част. Сортираната част се намира в лявата част на масива, а несортираната – в дясната. При всяка стъпка на алгоритъма, сортираната част се разширява надясно с един елемент, а несортираната – намалява с един от ляво.

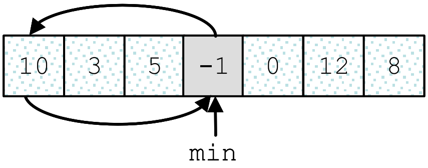
Нека разгледаме паралелно с обясненията един пример. Нека имаме следния несортиран масив от числа:



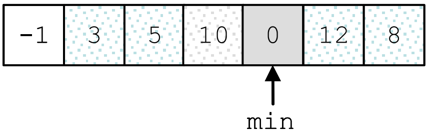
При всяка стъпка, нашият алгоритъм трябва да намери минималния елемент в несортираната част на масива:



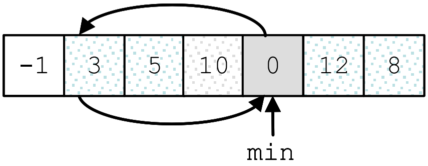
След това, трябва да размени намерения минимален елемент с първия елемент от несортираната част на масива:

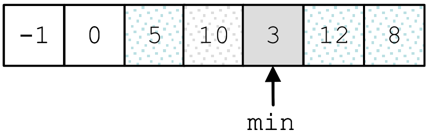


След което, отново се търси минималният елемент в оставащата несортирана част на масива (всички елементи без първия):

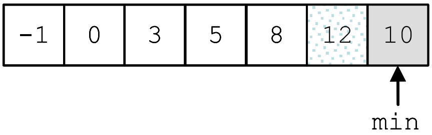


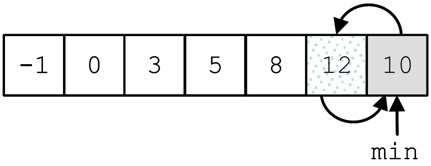
Тя се разменя с първия елемент от оставащата несортирана част:


-1
3
5
10
0
12
8
min

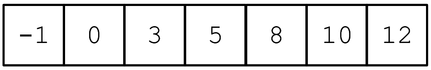

-1
0
5
10
3
12
8
min

Тази стъпка се повтаря, докато несортираната част на масива не бъде изчерпана:


-1
0
3
5
8
12
10
min


-1
0
3
5
8
12
10
min

Накрая масивът е сортиран:



Ето какъв вид добива нашия метод, след имплементацията на току-що описания алгоритъм (сортиране чрез пряка селекция):

```
static int[] Sort(params int[] numbers)
{
   // The sorting logic:
   for (int i = 0; i < numbers.Length - 1; i++)
   {
      // Loop operating over the unsorted part of the array
      for (int j = i + 1; j < numbers.Length; j++)
      {
         // Swapping the values
         if (numbers[i] > numbers[j])
         {
            int temp = numbers[i];
            numbers[i] = numbers[j];
            numbers[j] = temp;
         }
      }
   } // End of the sorting logic
   return numbers;
}
```

Нека декларираме и един метод **PrintNumbers(params int[])** за извеждане на списъка с числа в конзолата и тестваме с нашия примерен масив от числа като напишем няколко реда в **Main(…)** метода:

SortingEngine.cs

```
using System;

class SortingEngine
{
   static int[] Sort(params int[] numbers)
   {
      // The sorting logic:
      for (int i = 0; i < numbers.Length - 1; i++)
      {
         // Loop that is operating over the un-sorted part of
         // the array
         for (int j = i + 1; j < numbers.Length; j++)
         {
            // Swapping the values
            if (numbers[i] > numbers[j])
            {
               int oldNum = numbers[i];
               numbers[i] = numbers[j];
               numbers[j] = oldNum;
            }
         }
      } // End of the sorting logic
      return numbers;
   }

   static void PrintNumbers(params int[] numbers)
   {
      for (int i = 0; i < numbers.Length; i++)
      {
         Console.Write("{0}", numbers[i]);
         if (i < (numbers.Length - 1))
         {
            Console.Write(", ");
         }
      }
   }

   static void Main()
   {
      int[] numbers = Sort(10, 3, 5, -1, 0, 12, 8);
      PrintNumbers(numbers);
   }
}
```

Съответно, след компилирането и изпълнението на този код, резултатът е точно този, който очакваме – масивът е сортиран по големина в нарастващ ред:

```
-1, 0, 3, 5, 8, 10, 12
```

Утвърдени практики при работа с методи

Въпреки че в главата "Качествен програмен код" ще обясним повече за добрите практики при писане на методи, нека прегледаме още сега някои основни правила при работа с методи, които показват добър стил на програмиране:

- Всеки метод трябва да решава самостоятелна, добре дефинирана задача. Това свойство се нарича **strong cohesion**, т.е. фокусиране върху една, единствена задача, а не няколко несвързани задачи. Ако даден метод прави само едно нещо, кодът му е по-разбираем и по-лесен за поддръжка. Един метод не трябва да решава няколко задачи едновременно!
- Един метод трябва да има **добро име**, т.е. име, което описва какво прави той. Примерно метод, който сортира числа, трябва да се казва **`SortNumbers()`**, а не **`Number()`** или **`Processing()`** или **`Method2()`**. Ако не можете да измислите подходящо име за даден метод, то най-вероятно методът решава повече от една задача и трябва да се раздели на няколко отделни метода.
- Имената на методите е препоръчително да **изразяват действие**, поради което трябва да бъдат съставени от глагол или от глагол + съществително име (евентуално с прилагателно, което пояснява съществителното), примерно **`FindSmallestElement()`** или **`Sort(int[] arr)`** или **`ReadInputData()`**.
- Имената на методите в C# е прието да започват с голяма буква. Използва се правилото **PascalCase**, т.е. всяка нова дума, която се долепя в задната част на името на метода, започва с главна буква, например **`SendEmail(…)`**, а не **`sendEmail(…)`** или **`send_email(…)`**.
- Един метод или трябва да свърши работата, която е описана от името му, или трябва да съобщи за грешка. Не е коректно методите да връщат грешен или странен резултат при некоректни входни данни. **Методът или решава задачата, за която е предназначен, или връща грешка**. Всякакво друго поведение е некоректно. Ще обясним в детайли по какъв начин методите могат да съобщават за грешки в главата "Обработка на изключения".
- Един метод трябва да бъде минимално обвързан с обкръжаващата го среда (най-вече с класа, в който е дефиниран). Това означава, че

методът трябва да обработва данни, идващи като параметри, а не данни, достъпни по друг начин и не трябва да има странични ефекти (например да промени някоя глобално достъпна променлива). Това свойство на методите се нарича **loose coupling**.

- Препоръчва се методите **да бъдат кратки**. Трябва да се избягват методи, които са по-дълги от "един екран". За да се постигне това, логиката имплементирана в метода, се разделя по функционалност на няколко по-малки метода и след това тези методи се извикват в "дългия" до момента метод.
- За да се подобри четимостта и прегледността на кода, е добре функционалност, която е добре обособена логически, да се отделя в метод. Например, ако имаме метод за намиране на обема на язовир, процесът на пресмятане на обем на паралелепипед може да се дефинира в отделен метод и след това този нов метод да се извика многократно от метода, който пресмята обема на язовира. Така естествената **подзадача се отделя от основната задача**. Тъй като язовирът може да се разглежда като съставен от много на брой паралелепипеди, то изчисляването на обема на един от тях е логически обособена функционалност.

Упражнения

1. Напишете метод, който при подадено име отпечатва в конзолата "`Hello, <name>!`" (например "`Hello, Peter!`"). Напишете програма, която тества този метод дали работи правилно.
2. Създайте метод `GetMax()` с два целочислени (`int`) параметъра, който връща по-голямото от двете числа. Напишете програма, която прочита три цели числа от конзолата и отпечатва най-голямото от тях, използвайки метода `GetMax()`.
3. Напишете метод, който връща английското наименование на последната цифра от дадено число. Примери: за числото `512` отпечатва "`two`"; за числото `1024` – "`four`".
4. Напишете метод, който намира колко пъти дадено число се среща в даден масив. Напишете програма, която проверява дали този метод работи правилно.
5. Напишете метод, който проверява дали елемент, намиращ се на дадена позиция от масив, е по-голям, или съответно по-малък от двата му съседа. Тествайте метода дали работи коректно.
6. Напишете метод, който връща позицията на първия елемент на масив, който е по-голям от двата свои съседи едновременно, или `-1`, ако няма такъв елемент.
7. Напишете метод, който отпечатва цифрите на дадено десетично число в обратен ред. Например `256`, трябва да бъде отпечатано като `652`.

8. Напишете метод, който пресмята сумата на две цели положителни цели числа. Числата са представени като масив от цифрите си, като последната цифра (единиците) са записани в масива под индекс 0. Направете така, че метода да работи за числа с дължина до 10 000 цифри.

9. Напишете метод, който намира най-големия елемент в част от масив. Използвайте метода за да сортирате възходящо/низходящо даден масив.

10. Напишете програма, която пресмята и отпечатва **`n!`** за всяко n в интервала `[1…100]`.

11. Напишете програма, която решава следните задачи:

    - Обръща последователността на цифрите на едно число.
    - Пресмята средното аритметично на дадена поредица от числа.
    - Решава линейното уравнение **`a * x + b = 0`**.

    Създайте подходящи методи за всяка една от задачите.

    Напишете програмата така, че на потребителя да му бъде изведено текстово меню, от което да избира коя от задачите да решава.

    Направете проверка на входните данни:

    - Десетичното число трябва да е неотрицателно.
    - Редицата не трябва да е празна.
    - Коефициентът **`a`** не трябва да е `0`.

12. Напишете метод, който събира два полинома с цели коефициенти, например $(3x^2 + x - 3) + (x - 1) = (3x^2 + 2x - 4)$.

13. Напишете метод, който умножава два полинома с цели коефициенти, например $(3x^2 + x - 3) * (x - 1) = (3x^3 - 2x^2 - 4x + 3)$.

Решения и упътвания

1. Използвайте метод с параметър **`string`**. Ако ви е интересно, вместо да правите програма, която да тества дали даден метод работи коректно, можете да потърсите в Интернет информация за "unit testing" и да си напишете собствени unit тестове върху методите, които създавате. За всички добри софтуерни продукти се пишат unit тестове.

2. Използвайте свойството **`Max(a, b, c) = Max(Max(a, b), c)`**.

3. Използвайте остатъка при деление на 10 и **`switch`** конструкцията.

4. Методът трябва да приема като параметър масив от числа (**`int[]`**) и търсеното число (**`int`**).

5. Елементите на първа и последна позиция в масива, ще бъдат сравнявани съответно само с десния и левия си съсед.

6. Модифицирайте метода, имплементиран в предходната задача.

7. Има два начина:

   Първи начин: Нека числото е **`num`**. Докато **`num ≠ 0`** отпечатваме последната му цифра (**`num % 10`**) и след това разделяме **`num`** на 10.

   Втори начин: преобразуваме числото в **`string`** и го отпечатваме отзад напред чрез **`for`** цикъл.

8. Трябва да имплементирате собствен метод за умножение на големи числа. На нулева позиция в масива пазете единиците, на първа позиция – десетиците и т.н. Когато събирате 2 големи числа, единиците на сумата ще е **`(firstNumber[0] + secondNumber[0]) % 10`**, десетиците ще са равни на **`(firstNumber[1] + secondNumber[1]) % 10`** + **`(firstNumber[0] + secondNumber[0])/10`** и т.н.

9. Първо напишете метод, който намира максималния елемент в целия масив, и след това го модифицирайте да намира такъв елемент от даден интервал.

10. Трябва да имплементирате собствен метод за умножение на големи цели числа, тъй като **`100!`** не може да се събере в **`long`**. Можете да представите числата в масив в обратен ред, с по една цифра във всеки елемент. Например числото 512 може да се представи като {2, 1, 5}. След това умножението можете да реализирате, както сте учили в училище (умножавате цифра по цифра и събирате резултатите с отместване на разрядите).

    Друг, по-лесен вариант да работите с големи числа като 100!, е чрез библиотеката **`System.Numerics.dll`**, която можете да използвате в проектите си като преди това добавите референция към нея. Потърсете информация в Интернет как да използвате библиотеката и класът **`System.Numerics.BigInteger`**.

11. Създайте първо необходимите ви методи. Менюто реализирайте чрез извеждане на списък от номерирани действия (1 - обръщане, 2 - средно аритметично, 3 - уравнение) и избор на число между 1 и 3.

12. Използвайте масиви за представяне на многочлените и правилата за събиране, които познавате от математиката. Например многочленът **($3x^2$ `+ x - 3`)** може да се представи като масив от числата `[-3, 1, 3]`. Обърнете внимание, че е по-удачно на нулева позиция да поставим коефициентът пред x^0 (за нашия пример -3), на първа – коефициентът пред x^1 (за нашия пример 1) и т.н.

13. Използвайте упътването от предходната задача и правилата за умножение на полиноми от математиката.

Глава 10. Рекурсия

В тази тема...

В настоящата тема ще се запознаем с рекурсията и нейните приложения. Рекурсията представлява мощна техника, при която един метод извиква сам себе си. С нея могат да се решават сложни комбинаторни задачи, при които с лекота могат да бъдат изчерпвани различни комбинаторни конфигурации. Ще ви покажем много примери за правилно и неправилно използване на рекурсия и ще ви убедим колко полезна може да е тя.

Какво е рекурсия?

Един обект наричаме **рекурсивен**, ако съдържа себе си или е дефиниран чрез себе си.

Рекурсия е програмна техника, при която даден **метод извиква сам себе си** при решаването на определен проблем. Такива методи наричаме **рекурсивни**.

Рекурсията е програмна техника, чиято правилна употреба води до елегантни решения на определени проблеми. Понякога нейното използване може да опрости значително кода и да подобри четимостта му.

Пример за рекурсия

Нека разгледаме числата на Фибоначи. Това са членовете на следната редица:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Всеки член на редицата се получава като сума на предходните два. Първите два члена по дефиниция са равни на 1, т.е. в сила е:

$F_1 = F_2 = 1$

$F_i = F_{i-1} + F_{i-2}$ (за i > 2)

Изхождайки директно от дефиницията, можем да реализираме следния рекурсивен метод за намиране на n-тото число на Фибоначи:

```
static long Fib(int n)
{
   if (n <= 2)
   {
      return 1;
   }
   return Fib(n - 1) + Fib(n - 2);
}
```

Този пример ни показва, колко проста и естествена може да бъде реализацията на дадено решение с помощта на рекурсия.

От друга страна, той може да ни служи и като пример, колко трябва да сме внимателни при използването на рекурсия. Макар, че е интуитивно, текущото решение е един от класическите примери, когато използването на рекурсия е изключително неефективно, поради множеството излишни изчисления (на едни и същи членове на редицата) в следствие на рекурсивните извиквания.

На предимствата и недостатъците от използване на рекурсия, ще се спрем в детайли малко по-късно в настоящата тема.

Пряка и косвена рекурсия

Когато в тялото на метод се извършва извикване на същия метод, казваме, че методът е **пряко рекурсивен**.

Ако метод A се обръща към метод B, B към C, а C отново към A, казваме, че методът A, както и методите B и C са **непряко (косвено) рекурсивни** или **взаимно-рекурсивни**.

Веригата от извиквания при косвената рекурсия може да съдържа множество методи, както и разклонения, т.е. при наличие на едно условие да се извиква един метод, а при различно условие да се извиква друг.

Дъно на рекурсията

Реализирайки рекурсия, трябва да сме сигурни, че след краен брой стъпки ще получим конкретен резултат. Затова трябва да имаме един или няколко случаи, чието решение можем да намерим директно, без рекурсивно извикване. Тези случаи наричаме **дъно на рекурсията**.

В примера с числата на Фибоначи, дъното на рекурсията е случаят, когато `n` е по-малко или равно на 2. При него можем директно да върнем резултат, без да извършваме рекурсивни извиквания, тъй като по дефиниция първите два члена на редицата на Фибоначи са равни на 1.

Ако даден рекурсивен метод няма дъно на рекурсията, тя ще стане **безкрайна** и резултатът ще е `StackOverflowException`.

Създаване на рекурсивни методи

Когато създаваме рекурсивни методи, трябва разбием задачата, която решаваме, на подзадачи, за чието решение можем да използваме същия алгоритъм (рекурсивно).

Комбинирането на решенията на всички подзадачи, трябва да води до решение на изходната задача.

При всяко рекурсивно извикване, проблемната област трябва да се ограничава така, че в даден момент да бъде достигнато дъното на рекурсията, т.е. разбиването на всяка от подзадачите трябва да води рано или късно до дъното на рекурсията.

Рекурсивно изчисляване на факториел

Използването на рекурсия ще илюстрираме с един класически пример – рекурсивно изчисляване на факториел.

Факториел от n (записва се n!) е произведението на естествените числа от 1 до n, като по дефиниция 0! = 1.

$$n! = 1.2.3\ldots n$$

Рекурентна дефиниция

При създаването на нашето решение, много по-удобно е да използваме съответната рекурентна дефиниция на факториел:

n! = 1, при n = 0

n! = n.(n-1)! за n>0

Намиране на рекурентна зависимост

Наличието на рекурентна зависимост не винаги е очевидно. Понякога се налага сами да я открием. В нашия случай можем да направим това, анализирайки проблема и пресмятайки стойностите на факториел за първите няколко естествени числа.

```
0! = 1
1! = 1 = 1.1 = 1.0!
2! = 2.1 = 2.1!
3! = 3.2.1 = 3.2!
4! = 4.3.2.1 = 4.3!
5! = 5.4.3.2.1 = 5.4!
```

От тук лесно се вижда рекурентната зависимост:

```
n! = n.(n-1)!
```

Реализация на алгоритъма

Дъното на нашата рекурсия е най-простият случай n = 0, при който стойността на факториел е 1.

В останалите случаи, решаваме задачата за n-1 и умножаваме получения резултат по n. Така след краен брой стъпки, със сигурност ще достигнем дъното на рекурсията, понеже между 0 и n има краен брой естествени числа.

След като имаме налице тези ключови условия, можем да реализираме метод изчисляващ факториел:

```
static decimal Factorial(int n)
{
   // The bottom of the recursion
   if (n == 0)
   {
      return 1;
   }
   // Recursive call: the method calls itself
   else
   {
        return n * Factorial(n - 1);
    }
}
```

Използвайки този метод, можем да създадем приложение, което чете от конзолата цяло число, изчислява факториела му и отпечатва получената стойност:

RecursiveFactorial.cs

```
using System;

class RecursiveFactorial
{
    static void Main()
    {
        Console.Write("n = ");
        int n = int.Parse(Console.ReadLine());

        decimal factorial = Factorial(n);
        Console.WriteLine("{0}! = {1}", n, factorial);
    }

    static decimal Factorial(int n)
    {
        // The bottom of the recursion
        if (n == 0)
        {
            return 1;
        }
        // Recursive call: the method calls itself
        else
        {
            return n * Factorial(n - 1);
        }
    }
}
```

Ето какъв ще бъде резултатът от изпълнението на приложението, ако въведем 5 за стойност на n:

```
n = 5
5! = 120
```

Рекурсия или итерация

Изчислението на факториел често се дава като пример при обяснението на понятието рекурсия, но в този случай, както и в редица други, рекурсията далеч не е най-добрият подход.

Често, ако е зададена рекурентна дефиниция на проблема, рекурентното решение е интуитивно и не представлява трудност, докато **итеративно** (последователно) решение не винаги е очевидно.

В конкретния случай, реализацията на итеративно решение е също толкова кратка и проста, но малко по-ефективна:

```
static decimal Factorial(int n)
{
   decimal result = 1;

   for (int i = 1; i <= n; i++)
   {
      result = result * i;
   }

   return result;
}
```

Предимствата и недостатъците при използването на рекурсия и итерация ще разгледаме малко по-нататък в настоящата тема.

За момента трябва да запомним, че преди да пристъпим към реализацията на рекурсивно решение, трябва да помислим и за итеративен вариант, след което да изберем по-доброто решение според конкретната ситуация.

Нека се спрем на още един пример, където можем да използваме рекурсия за решаване на проблема, като ще разгледаме и итеративно решение.

Имитация на N вложени цикъла

Често се налага да пишем вложени цикли. Когато те са два, три или друг предварително известен брой, това става лесно. Ако броят им, обаче, не е предварително известен, се налага да търсим алтернативен подход. Такъв е случаят в следващата задача.

Да се напише програма, която симулира изпълнението на `N` вложени цикъла от `1` до `K`, където `N` и `K` се въвеждат от потребителя. Резултатът от изпълнението на програмата, трябва да е еквивалентен на изпълнението на следния фрагмент:

```
for (a1 = 1; a1 <= K; a1++)
   for (a2 = 1; a2 <= K; a2++)
      for (a3 = 1; a3 <= K; a3++)
         ...
         for (aN = 1; aN <= K; aN++)
            Console.WriteLine("{0} {1} {2} ... {N}",
               a1, a2, a3, ..., aN);
```

Например при N = 2 и K = 3 (което е еквивалентно на 2 вложени цикъла от 1 до 3) и при N = 3 и K = 3, резултатите трябва да са съответно:

```
                1 1                 1 1 1
                1 2                 1 1 2
                1 3                 1 1 3
  N = 2         2 1       N = 3     1 2 1
  K = 3   ->    2 2       K = 3 ->  ...
                2 3                 3 2 3
                3 1                 3 3 1
                3 2                 3 3 2
                3 3                 3 3 3
```

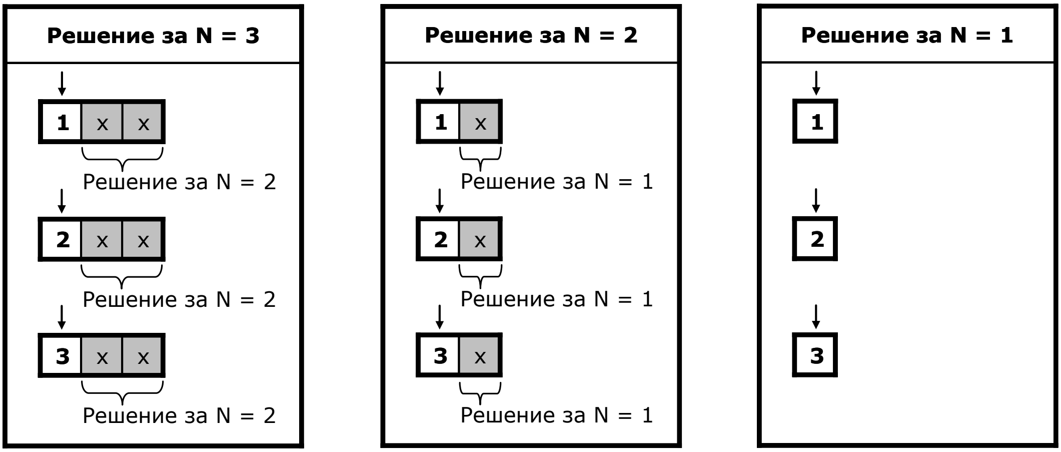
Алгоритъмът за решаване на тази задача не е така очевиден, както в предходния пример. Нека разгледаме две различни решения – едното рекурсивно, а другото – итеративно.

Всеки ред от резултата, можем да разглеждаме като наредена последователност от N числа. Първото число представлява текущата стойност на брояча на първия цикъл, второто на втория и т.н. На всяка една позиция, можем да имаме стойност между 1 и K. Решението на нашата задача се свежда до намирането на всички наредени N-торки за дадени N и K.

Вложени цикли – рекурсивен вариант

Първият проблем, който се изправя пред нас, ако търсим рекурсивен подход за решаване на тази задача, е намирането на рекурентна зависимост. Нека се вгледаме малко по-внимателно в примера от условието на задачата и да направим някои разсъждения.

Забелязваме, че ако сме пресметнали решението за N = 2, то решението за N = 3 можем да получим, като поставим на първа позиция всяка една от стойностите на K (в случая от 1 до 3), а на останалите 2 позиции поставяме последователно всяка от двойките числа, получени от решението за N = 2. Можем да проверим, че това правило важи и при стойности на N по-големи от 3.


Решение за N = 3
1 x x
Решение за N = 2
2 x x
Решение за N = 2
3 x x
Решение за N = 2
Решение за N = 2
1 x
Решение за N = 1
2 x
Решение за N = 1
3 x
Решение за N = 1
Решение за N = 1
1
2
3

Така получаваме следната зависимост – започвайки от първа позиция, поставяме на текущата позиция всяка една от стойностите от 1 до K и продължаваме рекурсивно със следващата позиция. Това продължава, докато достигнем позиция N, след което отпечатваме получения резултат. Ето как изглежда и съответният метод на C#:

```
static void NestedLoops(int currentLoop)
{
   if (currentLoop == numberOfLoops)
   {
      PrintLoops();
      return;
   }

   for (int counter=1; counter<=numberOfIterations; counter++)
   {
      loops[currentLoop] = counter;
      NestedLoops(currentLoop + 1);
   }
}
```

Последователността от стойности ще пазим в масив наречен **`loops`**, който при нужда ще бъде отпечатван от метода **`PrintLoops()`**.

Методът **`NestedLoops(…)`** има един параметър, указващ текущата позиция, на която ще поставяме стойности.

В цикъла, поставяме последователно на текущата позиция всяка една от възможните стойности (променливата **`numberOfIterations`** съдържа стойността на K въведена от потребителя), след което извикваме рекурсивно метода **`NestedLoops(…)`** за следващата позиция.

Дъното на рекурсията се достига, когато текущата позиция достигне N (променливата **`numberOfLoops`** съдържа стойността на N въведена от потребителя). В този момент имаме стойности на всички позиции и отпечатваме последователността.

Ето и цялостна реализация на решението:

RecursiveNestedLoops.cs

```
using System;

class RecursiveNestedLoops
{
   static int numberOfLoops;
   static int numberOfIterations;
   static int[] loops;

   static void Main()
   {
        Console.Write("N = ");
        numberOfLoops = int.Parse(Console.ReadLine());

        Console.Write("K = ");
        numberOfIterations = int.Parse(Console.ReadLine());

        loops = new int[numberOfLoops];

        NestedLoops(0);
    }

    static void NestedLoops(int currentLoop)
    {
        if (currentLoop == numberOfLoops)
        {
            PrintLoops();
            return;
        }

        for (int counter=1; counter<=numberOfIterations; counter++)
        {
            loops[currentLoop] = counter;
            NestedLoops(currentLoop + 1);
        }
    }

    static void PrintLoops()
    {
        for (int i = 0; i < numberOfLoops; i++)
        {
            Console.Write("{0} ", loops[i]);
        }
        Console.WriteLine();
    }
}
```

Ако стартираме приложението и въведем за стойности на **N** и **K** съответно 2 и 4, ще получим следния резултат:

```
N = 2
K = 4
1 1
1 2
1 3
1 4
2 1
2 2
2 3
2 4
3 1
```

```
3 2
3 3
3 4
4 1
4 2
4 3
4 4
```

В метода **`Main()`** въвеждаме стойности за **`N`** и **`K`**, създаваме масива, в който ще пазим последователността от стойности, след което извикваме метода **`NestedLoops(…)`**, започвайки от първа позиция.

Забележете, че като параметър на метода подаваме 0, понеже пазим последователността от стойности в масив, а както вече знаем, номерацията на елементите в масив започва от 0.

Методът **`PrintLoops()`** обхожда всички елементи на масива и ги отпечатва на конзолата.

Вложени цикли – итеративен вариант

За реализацията на итеративно решение, можем да използваме следния алгоритъм, който на всяка итерация намира следващата последователност от числа и я отпечатва:

1. В начално състояние на всички позиции поставяме числото 1.
2. Отпечатваме текущата последователност от числа.
3. Увеличаваме с единица числото намиращо се на позиция N. Ако получената стойност е по-голяма от K, заменяме я с 1 и увеличаваме с единица стойността на позиция N-1. Ако и нейната стойност е станала по-голяма от K, също я заменяме с 1 и увеличаваме с единица стойността на позиция N-2 и т.н.
4. Ако стойността на първа позиция, е станала по-голяма от K, алгоритъмът приключва работа.
5. Преминаваме към стъпка 2.

Следва примерна реализация на описания алгоритъм:

IterativeNestedLoops.cs

```
using System;

class IterativeNestedLoops
{
   static int numberOfLoops;
   static int numberOfIterations;
   static int[] loops;
static void Main()
{
   Console.Write("N = ");
   numberOfLoops = int.Parse(Console.ReadLine());

   Console.Write("K = ");
   numberOfIterations = int.Parse(Console.ReadLine());

   loops = new int[numberOfLoops];

   NestedLoops();
}

static void NestedLoops()
{
   InitLoops();

   int currentPosition;

   while (true)
   {
      PrintLoops();

      currentPosition = numberOfLoops - 1;
      loops[currentPosition] = loops[currentPosition] + 1;

      while (loops[currentPosition] > numberOfIterations)
      {
         loops[currentPosition] = 1;
         currentPosition--;

         if (currentPosition < 0)
         {
            return;
         }
         loops[currentPosition] = loops[currentPosition] + 1;
      }
   }
}

static void InitLoops()
{
   for (int i = 0; i < numberOfLoops; i++)
   {
      loops[i] = 1;
   }
}

static void PrintLoops()
{
        for (int i = 0; i < numberOfLoops; i++)
        {
            Console.Write("{0} ", loops[i]);
        }
        Console.WriteLine();
    }
}
```

Методите **`Main()`** и **`PrintLoops()`** са същите, както в реализацията на рекурсивното решение.

Различен е методът **`NestedLoops()`**, който сега реализира алгоритъма за итеративно решаване на проблема и поради това не приема параметър, както в рекурсивния вариант.

В самото начало на този метод извикваме метода **`InitLoops()`**, който обхожда елементите на масива и поставя на всички позиции единици.

Стъпките на алгоритъма реализираме в безкраен цикъл, от който ще излезем в подходящ момент, прекратявайки изпълнението на метода чрез оператора **`return`**.

Интересен е начинът, по който реализираме стъпка 3 от алгоритъма. Проверката за стойности по-големи от K, заменянето им с единица и увеличаването на стойността на предходна позиция (след което правим същата проверка и за нея), реализираме с помощта на един **`while`** цикъл, в който влизаме само, ако стойността е по-голяма от K.

За целта първо заменяме стойността на текущата позиция с единица. След това текуща става позицията преди нея. После увеличаваме стойността на новата позиция с единица и се връщаме в началото на цикъла. Тези действия продължават, докато стойността на текуща позиция не се окаже по-малка или равна на K (променливата **`numberOfIterations`** съдържа стойността на K), при което излизаме от цикъла.

В момента, когато на първа позиция стойността стане по-голяма от K (това е моментът, когато трябва да приключим изпълнението), на нейно място поставяме единица и опитваме да увеличим стойността на предходната позиция. В този момент стойността на променливата **`currentPosition`** става отрицателна (понеже първата позиция в масив е 0), при което прекратяваме изпълнението на метода чрез оператора **`return`**. С това задачата ни е изпълнена.

Можем да тестваме, примерно с N=3 и K=2:

```
N = 3
K = 2
1 1 1
1 1 2
1 2 1
1 2 2
```

```
2 1 1
2 1 2
2 2 1
2 2 2
```

Кога да използваме рекурсия и кога итерация?

Когато алгоритъмът за решаване на даден проблем е рекурсивен, реализирането на рекурсивно решение, може да бъде много по-четливо и елегантно от реализирането на итеративно решение на същия проблем.

Понякога дефинирането на еквивалентен итеративен алгоритъм е значително по-трудно и не е лесно да се докаже, че двата алгоритъма са еквивалентни.

В определени случаи, чрез използването на рекурсия, можем да постигнем много по-прости, кратки и лесни за разбиране решения.

От друга страна, рекурсивните извиквания, може да консумират много повече ресурси и памет. При всяко рекурсивно извикване, в стека се заделя нова памет за аргументите, локалните променливи и връщаните резултати. При прекалено много рекурсивни извиквания може да се получи препълване на стека, поради недостиг на памет.

В дадени ситуации рекурсивните решения може да са много по-трудни за разбиране и проследяване от съответните итеративни решения.

Рекурсията е мощна програмна техника, но трябва внимателно да преценяваме, преди да я използваме. При неправилна употреба, тя може да доведе до неефективни и трудни за разбиране и поддръжка решения.

Ако чрез използването на рекурсия, постигаме по-просто, кратко и по-лесно за разбиране решение, като това не е за сметка на ефективността и не предизвиква други странични ефекти, тогава можем да предпочетем рекурсивното решение. В противен случай, е добре да помислим дали не е по-подходящо да използваме итерация.

Числа на Фибоначи – неефективна рекурсия

Нека се върнем отново към примера с намирането на n-тото число на Фибоначи и да разгледаме по-подробно рекурсивното решение:

```
static long Fib(int n)
{
   if (n <= 2)
   {
      return 1;
   }
   return Fib(n - 1) + Fib(n - 2);
}
```

Това решение е интуитивно, кратко и лесно за разбиране. На пръв поглед изглежда, че това е чудесен пример за приложение на рекурсията. Истината е, че това е един от класическите примери за неподходящо използване на рекурсия. Нека стартираме следното приложение:

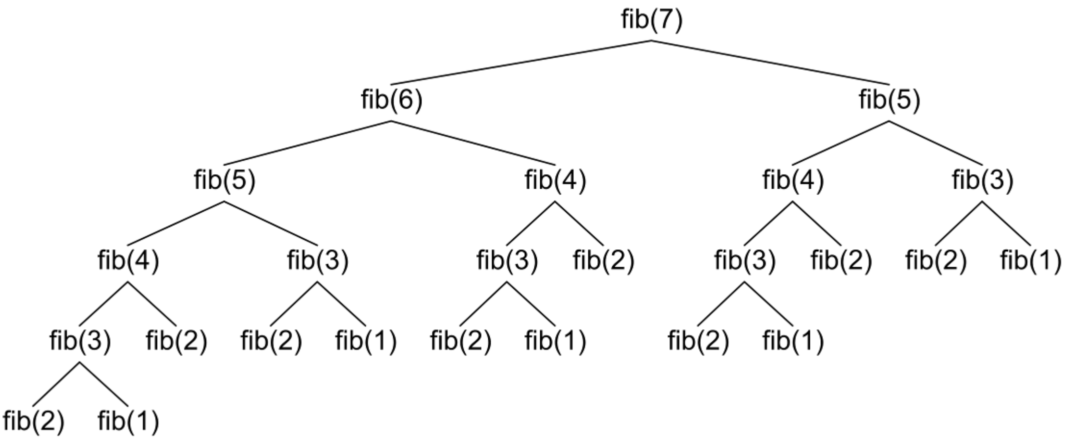
RecursiveFibonacci.cs

```
using System;

class RecursiveFibonacci
{
   static void Main()
   {
      Console.Write("n = ");
      int n = int.Parse(Console.ReadLine());

      long result = Fib(n);
      Console.WriteLine("fib({0}) = {1}", n, result);
   }

   static long Fib(int n)
   {
      if (n <= 2)
      {
         return 1;
      }
      return Fib(n - 1) + Fib(n - 2);
   }
}
```

Ако зададем като стойност n = 100, изчисленията ще отнемат толкова дълго време, че едва ли някой ще изчака, за да види резултата. Причината за това е, че подобна реализация е изключително неефективна. Всяко рекурсивно извикване води след себе си още две, при което дървото на извикванията расте експоненциално, както е показано на фигурата по-долу.

Броят на стъпките за изчисление на fib(100) е от порядъка на 1.6 на степен 100 (това се доказва математически), докато при линейно решение е само 100.

Проблемът произлиза от това, че се правят напълно излишни изчисления. Повечето членове на редицата се пресмятат многократно. Може да обърнете внимание колко много пъти на фигурата се среща fib(2).


fib(7)
fib(6)
fib(5)
fib(5)
fib(4)
fib(4)
fib(3)
fib(4)
fib(3)
fib(3)
fib(2)
fib(3)
fib(2)
fib(2)
fib(1)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
fib(2)
fib(1)
fib(2)
fib(1)

Числа на Фибоначи – ефективна рекурсия

Можем да оптимизираме рекурсивния метод за изчисление на числата на Фибоначи, като записваме вече пресметнатите числа в масив и извършваме рекурсивно извикване само ако числото, което пресмятаме, не е било вече пресметнато до момента. Благодарение на тази малка оптимизационна техника (известна в компютърните науки и в динамичното оптимиране с термина **memorization**), рекурсивното решение ще работи за линеен брой стъпки. Ето примерна реализация:

RecursiveFibonacciMemoization.cs

```
using System;

class RecursiveFibonacciMemoization
{
  static long[] numbers;

  static void Main()
  {
    Console.Write("n = ");
    int n = int.Parse(Console.ReadLine());

    numbers = new long[n + 2];
    numbers[1] = 1;
    numbers[2] = 1;

    long result = Fib(n);
    Console.WriteLine("fib({0}) = {1}", n, result);
  }

  static long Fib(int n)
  {
    if (0 == numbers[n])
    {
            numbers[n] = Fib(n - 1) + Fib(n - 2);
        }

        return numbers[n];
    }
}
```

Забелязвате ли разликата? Докато при първоначалния вариант, при n = 100, ни се струва, че изчисленията продължават безкрайно дълго, а при оптимизираното решение, получаваме отговор мигновено:

```
n = 100
fib(100) = 3736710778780434371
```

Числа на Фибоначи – итеративно решение

Не е трудно да забележим, че можем да решим проблема и без използването на рекурсия, пресмятайки числата на Фибоначи последователно. За целта ще пазим само последните два пресметнати члена на редицата и чрез тях ще получаваме следващия. Следва реализация на итеративния алгоритъм:

IterativeFibonacci.cs

```
using System;

class IterativeFibonacci
{
    static void Main()
    {
        Console.Write("n = ");
        int n = int.Parse(Console.ReadLine());

        long result = Fib(n);
        Console.WriteLine("fib({0}) = {1}", n, result);
    }

    static long Fib(int n)
    {
        long fn = 1;
        long fnMinus1 = 1;
        long fnMinus2 = 1;

        for (int i = 2; i < n; i++)
        {
            fn = fnMinus1 + fnMinus2;

            fnMinus2 = fnMinus1;
            fnMinus1 = fn;
    }

    return fn;
  }
}
```

Това решение е също толкова кратко и елегантно, но не крие рисковете от използването на рекурсия. Освен това то е ефективно и не изисква допълнителна памет.

Изхождайки от горните примери, можем да дадем следната препоръка:

Избягвайте рекурсията, освен, ако не сте сигурни как работи тя и какво точно се случва зад кулисите. Рекурсията е голямо и мощно оръжие, с което лесно можете да се застреляте в крака. Ползвайте я внимателно!

Ако следваме това правило, ще намалим значително вероятността за неправилно използване на рекурсия и последствията, произтичащи от него.

Още за рекурсията и итерацията

По принцип, когато имаме **линеен изчислителен процес**, не трябва да използваме рекурсия, защото итерацията може да се реализира изключително лесно и води до прости и ефективни изчисления. Пример за линеен изчислителен процес е изчислението на факториел. При него изчисляваме членовете на редица, в която всеки следващ член зависи единствено от предходните.

Линейните изчислителни процеси се характеризират с това, че на всяка стъпка от изчисленията рекурсията се извиква еднократно, само в една посока. Схематично линейният изчислителен процес можем да опишем така:

```
void Recursion(parameters)
{
  do some calculations;
  Recursion(some parameters);
  do some calculations;
}
```

При такъв процес, когато имаме само едно рекурсивно извикване в тялото на рекурсивния метод, не е нужно да ползваме рекурсия, защото итерацията е очевидна.

Понякога, обаче имаме разклонен или **дървовиден изчислителен процес**. Например имитацията на N вложени цикъла не може лесно да се замени с итерация. Вероятно сте забелязали, че нашият итеративен алгоритъм, който имитира вложените цикли, работи на абсолютно различен принцип. Опитайте да реализирате същото поведение без рекурсия и ще се убедите, че не е лесно.

По принцип всяка рекурсия може да се сведе до итерация чрез използване на стек на извикванията (какъвто се създава по време на изпълнение на програмата), но това е сложно и от него няма никаква полза. Рекурсията трябва да се ползва, когато дава просто, лесно за разбиране и ефективно решение на даден проблем, за който няма очевидно итеративно решение.

При дървовидните изчислителни процеси на всяка стъпка от рекурсията, се извършват няколко на брой рекурсивни извиквания и схемата на извършване на изчисленията може да се визуализира като дърво (а не като списък, както при линейните изчисления). Например при изчислението на числата на Фибоначи видяхме какво дърво на рекурсивните извиквания се получава.

Типичната схема на дървовидния изчислителен процес можем да опишем чрез псевдокод така:

```
void Recursion(parameters)
{
   do some calculations;
   Recursion(some parameters);
   ...
   Recursion(some other parameters);
   do some calculations;
}
```

Дървовидните изчислителни процеси не могат директно да бъдат сведени до рекурсивни (за разлика от линейните). Случаят с числата на Фибоначи е простичък, защото всяко следващо число се изчислява чрез предходните, които можем да изчислим предварително. Понякога, обаче всяко следващо число се изчислява не само чрез предходните, а и чрез следващите и рекурсивната зависимост не е толкова проста. В такъв случай рекурсията се оказва особено ефективна.

Ще илюстрираме последното твърдение с един класически пример.

Търсене на пътища в лабиринт – пример

Даден е лабиринт, който има правоъгълна форма и се състои от N*M квадратчета. Всяко квадратче е или проходимо, или не е проходимо. Търсач на приключения влиза в лабиринта от горния му ляв ъгъл (там е входът) и трябва да стигне до долния десен ъгъл на лабиринта (там е изходът). Търсачът на приключения може на всеки ход да се премести с една позиция нагоре, надолу, наляво или надясно, като няма право да излиза извън границите на от лабиринта и няма право да стъпва върху непроходими квадратчета. Преминаването през една и съща позиция повече от веднъж също е забранено (счита се, че търсачът на приключения се е загубил, ако се върне след няколко хода на място, където вече е бил). Да се напише компютърна програма, която отпечатва всички възможни пътища от началото до края на лабиринта.

Това е типичен пример за задача, която може лесно да се реши с рекурсия, докато с итерация решението е по-сложно и по-трудно за реализация.

Нека първо си нарисуваме един пример, за да си представим условието на задачата и да помислим за решение:

s			▨			
▨	▨		▨		▨	
	▨	▨	▨	▨	▨	
						e

Видно е, че има 3 различни пътя от началната позиция до крайната, които отговарят на изискванията на задачата (движение само по празни квадратчета и без преминаване по два пъти през никое от тях). Ето как изглеждат въпросните 3 пътя:

s	1	2	▨			
▨	▨	3	▨		▨	
6	5	4				
7	▨	▨	▨	▨	▨	
8	9	10	11	12	13	14

s	1	2	▨	8	9	10
▨	▨	3	▨	7	▨	11
		4	5	6		12
	▨	▨	▨	▨	▨	13
						14

s	1	2	▨			
▨	▨	3	▨		▨	
		4	5	6	7	8
	▨	▨	▨	▨	▨	9
						10

На фигурата по-горе с числата от 1 до 14 е означен номерът на съответната стъпка от пътя.

Пътища в лабиринт – рекурсивен алгоритъм

Как да решим задачата? Можем да разгледаме търсенето на дадена позиция в лабиринта до края на лабиринта като рекурсивен процес по следния начин:

- Нека текущата позиция в лабиринта е (row, col). В началото тръгваме от стартовата позиция (0,0).
- Ако текущата позиция е търсената позиция (N-1, M-1), то сме намерили път и трябва да го отпечатаме.
- Ако текущата позиция е непроходима, връщаме се назад (нямаме право да стъпваме в нея).
- Ако текущата позиция е вече посетена, връщаме се назад (нямаме право да стъпваме втори път в нея).

- В противен случай търсим път в четирите възможни посоки. Търсим рекурсивно (със същия алгоритъм) път към изхода на лабиринта като опитваме да ходим във всички възможни посоки:
    - Опитваме наляво: позиция (row, col-1).
    - Опитваме нагоре: позиция (row-1, col).
    - Опитваме надясно: позиция (row, col+1).
    - Опитваме надолу: позиция (row+1, col).

За да стигнем до този алгоритъм, разсъждаваме рекурсивно. Имаме задачата "търсене на път от дадена позиция до изхода". Тя може да се сведе до 4 подзадачи:

- търсене на път от позицията вляво от текущата до изхода;
- търсене на път от позицията нагоре от текущата до изхода;
- търсене на път от позицията вдясно от текущата до изхода;
- търсене на път от позицията надолу от текущата до изхода.

Ако от всяка възможна позиция, до която достигнем, проверим четирите възможни посоки и не се въртим в кръг (избягваме преминаване през позиция, на която вече сме били), би трябвало рано или късно да намерим изхода (ако съществува път към него).

Този път рекурсията не е толкова проста, както при предните задачи. На всяка стъпка трябва да проверим дали не сме стигнали изхода и дали не стъпваме в забранена позиция, след това трябва да отбележим позицията като посетена и да извикаме рекурсивното търсене на път в четирите посоки. След връщане от рекурсивните извиквания, трябва да отбележим обратно като непосетена позицията, от която се оттегляме. Такова обхождане е известно в информатиката като **търсене с връщане назад (backtracking)**.

Пътища в лабиринт – имплементация

За реализацията на алгоритъма ще ни е необходимо представяне на лабиринта. Ще ползваме двумерен масив от символи, като в него ще означим със символа ' ' (интервал) проходимите позиции, с '**e**' изхода от лабиринта и с '*****' непроходимите полета. Стартовата позиция ще означим като празна. Позициите, през които сме минали, ще означим със символа '**s**'. Ето как ще изглежда дефиницията на лабиринта за нашия пример:

```
static char[,] lab =
{
   {' ', ' ', ' ', '*', ' ', ' ', ' '},
   {'*', '*', ' ', '*', ' ', '*', ' '},
   {' ', ' ', ' ', ' ', ' ', ' ', ' '},
   {' ', '*', '*', '*', '*', '*', ' '},
   {' ', ' ', ' ', ' ', ' ', ' ', 'e'},
};
```

Нека се опитаме да реализираме рекурсивния метод за търсене в лабиринт. Той трябва да бъде нещо такова:

```
static char[,] lab =
{
   {' ', ' ', ' ', '*', ' ', ' ', ' '},
   {'*', '*', ' ', '*', ' ', '*', ' '},
   {' ', ' ', ' ', ' ', ' ', ' ', ' '},
   {' ', '*', '*', '*', '*', '*', ' '},
   {' ', ' ', ' ', ' ', ' ', ' ', 'e'},
};

static void FindPath(int row, int col)
{
   if ((col < 0) || (row < 0) ||
      (col >= lab.GetLength(1)) || (row >= lab.GetLength(0)))
   {
      // We are out of the labyrinth
      return;
   }

   // Check if we have found the exit
   if (lab[row, col] == 'e')
   {
      Console.WriteLine("Found the exit!");
   }

   if (lab[row, col] != ' ')
   {
      // The current cell is not free
      return;
   }

   // Mark the current cell as visited
   lab[row, col] = 's';

   // Invoke recursion to explore all possible directions
   FindPath(row, col - 1); // left
   FindPath(row - 1, col); // up
   FindPath(row, col + 1); // right
   FindPath(row + 1, col); // down

   // Mark back the current cell as free
   lab[row, col] = ' ';
}

static void Main()
```

```
{
  FindPath(0, 0);
}
```

Имплементацията стриктно следва описанието, дадено по-горе. В случая размерът на лабиринта не е записан в променливи N и M, а се извлича от двумерния масив **`lab`**, съхраняващ лабиринта: броят колони е **`lab.GetLength(1)`**, а броят редове е **`lab.GetLength(0)`**.

При влизане в рекурсивния метод за търсене първо се проверява дали няма излизане извън лабиринта. Ако има, търсенето от текущата позиция нататък се прекратява, защото е забранено излизане извън границите на лабиринта.

След това се проверява дали не сме намерили изхода. Ако сме го намерили, се отпечатва подходящо съобщение и търсенето от текущата позиция нататък приключва.

След това се проверява дали е свободна текущата клетка. Клетката е свободна, ако е проходима и не сме били на нея при някоя от предните стъпки (ако не е част от текущия път от стартовата позиция до текущата клетка на лабиринта).

При свободна клетка, се осъществява стъпване в нея. Това се извършва като се означи клетката като заета (със символа '**`s`**'). След това рекурсивно се търси път в четирите възможни посоки. След връщане от рекурсивното проучване на четирите възможни посоки, се отстъпва назад от текущата клетка и тя се маркира отново като свободна (връщане назад).

Маркирането на текущата клетка като свободна при излизане от рекурсията е важно, защото при връщане назад тя вече не е част от текущия път. Ако бъде пропуснато това действие, няма да бъдат намерени всички пътища до изхода, а само някои от тях.

Така изглежда рекурсивният метод за търсене на изхода в лабиринта. Остава само да го извикаме от **`Main()`** метода, започвайки търсенето на пътя от началната позиция (0, 0).

Ако стартираме програмата, ще видим следния резултат:

```
Found the exit!
Found the exit!
Found the exit!
```

Вижда се, че изходът е бил намерен точно 3 пъти. Изглежда алгоритъмът работи коректно. Липсва ни обаче отпечатването на самия път като последователност от позиции.

Пътища в лабиринт – запазване на пътищата

За да можем да отпечатаме пътищата, които намираме с нашия рекурсивен алгоритъм, можем да използваме масив, в който при всяко придвижване пазим посоката, която сме поели (**`L`** – наляво, **`U`** – нагоре, **`R`** – надясно, **`D`** – надолу). Този масив ще съдържа във всеки един момент текущия път от началото на лабиринта до текущата позиция.

Ще ни трябва един масив от символи и един брояч на стъпките, които сме направили. Броячът ще пази колко пъти сме се придвижили към следваща позиция рекурсивно, т.е. текущата дълбочина на рекурсията.

За да работи всичко коректно, е необходимо преди влизане в рекурсия да увеличаваме брояча и да запазваме посоката, която сме поели в текущата позиция от масива, а при връщане от рекурсията – да намаляваме брояча. При намиране на изхода можем да отпечатаме пътя – всички символи от масива от 0 до позицията, която броячът сочи.

Колко голям да бъде масивът? Отговорът на този въпрос е лесен; понеже в една клетка можем да влезем най-много веднъж, то никога пътят няма да е по-дълъг от общия брой клетки в лабиринта (N*M). В нашия случай размерът е 7*5, т.е. масивът е достатъчно да има 35 позиции.

Следва една примерна имплементация на описаната идея:

```
static char[,] lab =
{
  {' ', ' ', ' ', '*', ' ', ' ', ' '},
  {'*', '*', ' ', '*', ' ', '*', ' '},
  {' ', ' ', ' ', ' ', ' ', ' ', ' '},
  {' ', '*', '*', '*', '*', '*', ' '},
  {' ', ' ', ' ', ' ', ' ', ' ', 'e'},
};

static char[] path =
  new char[lab.GetLength(0) * lab.GetLength(1)];
static int position = 0;

static void FindPath(int row, int col, char direction)
{
  if ((col < 0) || (row < 0) ||
     (col >= lab.GetLength(1)) || (row >= lab.GetLength(0)))
  {
    // We are out of the labyrinth
    return;
  }

  // Append the direction to the path
  path[position] = direction;
  position++;
  // Check if we have found the exit
  if (lab[row, col] == 'e')
  {
    PrintPath(path, 1, position - 1);
  }

  if (lab[row, col] == ' ')
  {
    // The current cell is free. Mark it as visited
    lab[row, col] = 's';

    // Invoke recursion to explore all possible directions
    FindPath(row, col - 1, 'L'); // left
    FindPath(row - 1, col, 'U'); // up
    FindPath(row, col + 1, 'R'); // right
    FindPath(row + 1, col, 'D'); // down

    // Mark back the current cell as free
    lab[row, col] = ' ';
  }

  // Remove the direction from the path
  position--;
}

static void PrintPath(
  char[] path, int startPos, int endPos)
{
  Console.Write("Found path to the exit: ");
  for (int pos = startPos; pos <= endPos; pos++)
  {
    Console.Write(path[pos]);
  }
  Console.WriteLine();
}

static void Main()
{
  FindPath(0, 0, 'S');
}
```

За леснота добавихме още един параметър на рекурсивния метод за търсене на път до изхода от лабиринта: посоката, в която сме поели, за да дойдем на текущата позиция. Този параметър няма смисъл при първоначалното започване от стартовата позиция и затова в началото слагаме за посока някаква безсмислена стойност '`S`'. След това при отпечатването пропускаме първия елемент от пътя.

Ако стартираме програмата, ще получим трите възможни пътя от началото до края на лабиринта:

```
Found path to the exit: RRDDLLDDRRRRRR
Found path to the exit: RRDDRRUURRDDDD
Found path to the exit: RRDDRRRRDD
```

Пътища в лабиринт – тестване на програмата

Изглежда алгоритъмът работи. Остава да го тестваме с още малко примери, за да се убедим, че не сме допуснали някоя глупава грешка. Може да пробваме примерно с празен лабиринт с размер 1 на 1, с празен лабиринт с размер 3 на 3 и примерно с лабиринт, в който не съществува път до изхода, и накрая с огромен лабиринт, където пътищата са наистина много.

Ако изпълним тестовете, ще се убедим, че във всеки от тези необичайни случаи програмата работи коректно.

Примерен вход (лабиринт 1 на 1):

```
static char[,] lab =
{
   {'e'},
};
```

Примерен изход:

```
Found path to the exit:
```

Вижда се, че изходът е коректен, но пътят е празен (с дължина 0), тъй като стартовата позиция съвпада с изхода. Бихме могли да подобрим визуализацията в този случай (примерно да отпечатваме "**`Empty path`**").

Примерен вход (празен лабиринт 3 на 3):

```
static char[,] lab =
{
   {' ', ' ', ' '},
   {' ', ' ', ' '},
   {' ', ' ', 'e'},
};
```

Примерен изход:

```
Found path to the exit: RRDLLDRR
Found path to the exit: RRDLDR
Found path to the exit: RRDD
Found path to the exit: RDLDRR
Found path to the exit: RDRD
Found path to the exit: RDDR
Found path to the exit: DRURDD
Found path to the exit: DRRD
Found path to the exit: DRDR
Found path to the exit: DDRUURDD
Found path to the exit: DDRURD
Found path to the exit: DDRR
```

Вижда се, че изходът е коректен – това са всички пътища до изхода.

Примерен вход (лабиринт 5 на 3 без път до изхода):

```
static char[,] lab =
{
   {' ', '*', '*', ' ', ' '},
   {' ', ' ', ' ', '*', ' '},
   {'*', ' ', ' ', '*', 'e'},
};
```

Примерен изход:

```
(няма изход)
```

Вижда се, че изходът е коректен, но отново бихме могли да добавим по-приятелско съобщения (примерно "**No exit!**") вместо липса на какъвто и да е изход.

Сега остана да проверим какво се случва, когато имаме голям лабиринт. Ето примерен вход (лабиринт с размер 15 на 9):

```
static char[,] lab =
{
   {' ','*',' ',' ',' ',' ','*',' ',' ',' ',' ','*','*',' ',' '},
   {' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ','*','*','*',' ','*',' ',' ',' ',' ',' ','*','*','*','*'},
   {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ','e'},
};
```

Стартираме програмата и тя започва да печата непрекъснато пътища до изхода, но не свършва, защото пътищата са прекалено много. Ето как изглежда една малка част от изхода:

```
Found path to the exit:
DRDLDRRURUURRDLDRRURURRRDLLDLDRRURRURRURDDLLDLLDLLLDRRDLDRDRRURDRR
Found path to the exit:
DRDLDRRURUURRDLDRRURURRRDLLDLDRRURRURRURDDLLDLLDLLLDRRDLDRDRRRURRD
Found path to the exit:
DRDLDRRURUURRDLDRRURURRRDLLDLDRRURRURRURDDLLDLLDLLLDRRDLDRDRRRURDR
```

```
...
```

Сега, нека пробваме един последен пример – лабиринт с голям размер (15 на 9, в който не съществува път до изхода:

```
static char[,] lab =
{
   {' ','*',' ',' ',' ',' ','*',' ',' ',' ',' ','*','*',' ',' '},
   {' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
   {' ','*','*','*',' ','*',' ',' ',' ',' ',' ','*','*','*','*'},
   {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ','*','*',' ',' '},
   {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ','*','e'},
};
```

Стартираме програмата и тя заспива, без да отпечата нищо. Всъщност работи прекалено дълго, за да я изчакаме. Изглежда имаме проблем.

Какъв е проблемът? Проблемът е, че възможните пътища, които алгоритъмът анализира, са прекалено много и изследването им отнема прекалено много време. Да помислим колко се тези пътища. Ако средно един път до изхода е 20 стъпки и ако на всяка стъпка имаме 4 възможни посоки за продължение, то би трябвало да анализираме 4^{20} възможни пътя, което е ужасно голямо число. Тази оценка на броя възможности е изключително неточна, но дава ориентация за какъв порядък възможности става дума.

Какъв е изводът? Изводът е, че методът "**търсене с връщане назад**" (**backtracking**) не работи, когато вариантите са прекалено много, а фактът, че са прекалено много лесно може да се установи.

Няма да ви мъчим с опити да измислите решение на задачата. Проблемът за намиране на всички пътища в лабиринт няма ефективно решение при големи лабиринти.

Задачата има ефективно решение, ако бъде формулирана по друг начин: да се намери поне един изход от лабиринта. Тази задача е далеч по-лесна и може да се реши с една много малка промяна в примерния код: при връщане от рекурсията текущата позиция да не се маркира обратно като свободна. Това означава да изтрием следните редове код:

```
// Mark back the current cell as free
lab[row, col] = ' ';
```

Можем да се убедим, че след тази промяна, програмата много бързо установява, ако в лабиринта няма път до изхода, а ако има – много бързо намира един от пътищата (произволен).

Използване на рекурсия – изводи

Какъв е генералният извод от задачата за търсене на път в лабиринт? Изводът вече го формулирахме: ако не разбирате как работи рекурсията, избягвайте да я ползвате! Внимавайте, когато пишете рекурсивен код. Рекурсията е много мощен метод за решаване на комбинаторни задачи (задачи, в които изчерпваме варианти), но не е за всеки. Можете много лесно да сгрешите. Лесно можете да накарате програмата да "зависне" или да препълните стека с бездънна рекурсия. Винаги търсете итеративните решения, освен, ако не разбирате в голяма дълбочина как да ползвате рекурсията!

Колкото до задачата за търсене на най-къс път в лабиринт, можете да я решите елегантно без рекурсия с т.нар. **метод на вълната**, известен още като BFS (breadth-first search), който се реализира елементарно с една опашка. Повече за алгоритъма "BFS" можете да прочетете на неговата страница в Уикипедия: http://en.wikipedia.org/wiki/Breadth-first_search.

Упражнения

1. Напишете програма, която симулира изпълнението на `n` вложени цикъла от 1 до `n`. Пример:

```
                                              1 1 1
                                              1 1 2
                                              1 1 3
              1 1                             1 2 1
n=2 ->        1 2              n=3 ->         ….
              2 1                             3 2 3
              2 2                             3 3 1
                                              3 3 2
                                              3 3 3
```

2. Напишете рекурсивна програма, която генерира и отпечатва всички комбинации с повторение на `k` елемента над `n`-елементно множество.

   Примерен вход:

```
n = 3
k = 2
```

   Примерен изход:

```
(1 1), (1 2), (1 3), (2 2), (2 3), (3 3)
```

   Измислете и реализирайте итеративен алгоритъм за същата задача.

3. Напишете рекурсивна програма, която генерира всички вариации с повторение на **`n`** елемента от **`k`**-ти клас.

   Примерен вход:

   ```
   n = 3
   к = 2
   ```

   Примерен изход:

   ```
   (1 1), (1 2), (1 3), (2 1), (2 2), (2 3), (3 1), (3 2), (3 3)
   ```

   Измислете и реализирайте итеративен алгоритъм за същата задача.

4. Нека е дадено множество от символни низове. Да се напише рекурсивна програма, която генерира всички подмножества съставени от точно **`k`** на брой символни низа, избрани измежду елементите на това множество.

   Примерен вход:

   ```
   strings = {'test', 'rock', 'fun'}
   k = 2
   ```

   Примерен изход:

   ```
   (test rock), (test fun), (rock fun)
   ```

   Измислете и реализирайте итеративен алгоритъм за същата задача.

5. Напишете рекурсивна програма, която отпечатва всички подмножества на дадено множество от думи.

   Примерен вход:

   ```
   words = {'test', 'rock', 'fun'}
   ```

   Примерен изход:

   ```
   (), (test), (rock), (fun), (test rock), (test fun),
   (rock fun), (test rock fun)
   ```

   Измислете и реализирайте итеративен алгоритъм за същата задача.

6. Реализирайте алгоритъма "сортиране чрез сливане" (merge-sort). При него началният масив се разделя на две равни по големина части, които се сортират (рекурсивно чрез merge-sort) и след това двете сортирани части се сливат, за да се получи целият масив в сортиран вид.

7. Напишете рекурсивна програма, която генерира и отпечатва пермутациите на числата **`1, 2, …, n`**, за дадено цяло число **`n`**.

Примерен вход:

```
n = 3
```

Примерен изход:

```
(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)
```

8. Даден е масив с цели числа и число N. Напишете рекурсивна програма, която намира всички подмножества от числа от масива, които имат сума **`N`**. Например ако имаме масива `{2, 3, 1, -1}` и **`N=4`**, можем да получим **`N=4`** като сума по следните два начина: **`4=2+3-1; 4=3+1`**.
9. Даден е масив с цели **положителни** числа. Напишете програма, която проверява дали в масива съществуват едно или повече числа, чиято сума е **`N`**. Можете ли да решите задачата без рекурсия?
10. Дадена е матрица с проходими и непроходими клетки. Напишете рекурсивна програма, която намира всички пътища между две клетки в матрицата.
11. Модифицирайте горната програма, за да проверява дали съществува път между две клетки без да се намират всички възможни пътища. Тествайте за матрица 100x100 пълна само с проходими клетки.
12. Напишете програма, която намира най-дългата поредица от съседни проходими клетки в матрица.
13. Даден е двумерен масив с проходими и непроходими клетки. Напишете програма, която намира всички площи съставени само от проходими клетки.
14. Реализирайте алгоритъма BFS (breath-first search) за търсене на най-кратък път в лабиринт. Ако се затруднявате, потърсете информация в Интернет.
15. Напишете рекурсивна програма, която обхожда целия твърд диск `C:\` рекурсивно и отпечатва всички папки и файловете в тях.

Решения и упътвания

1. Направете метод, в който има цикъл и за всяко завъртане на цикъла да се вика същия метод.
2. И за рекурсивния и за итеративния вариант на задачата използвайте модификация на алгоритмите за генериране на **`N`** вложени цикли.
3. И за рекурсивния и за итеративния вариант на задачата използвайте модификация на алгоритмите за генериране на **`N`** вложени цикли.
4. Нека низовете са **`N`** на брой. Използвайте имитация на **`k`** вложени цикли (рекурсивна или итеративна). Трябва да генерирате всички множества от **`k`** елемента в диапазона `[0...N-1]`. За всяко такова

множество разглеждате числата от него като индекси в масива със символните низове и отпечатвате за всяко число съответния низ. За горния пример множеството `{0, 2}` означава нулевата и втората дума, т.е. `(test, fun)`.

5. Можете да използвате предходната задача и да я извикате N пъти, за да генерирате последователно празното множество (`k`=0), следвано от всички подмножества с 1 елемент (`k`=1), всички подмножества с 2 елемента (`k`=2), всички подмножества с 3 елемента (`k`=3) и т.н.

   Задачата има и много по-хитро решение: завъртате цикъл от 0 до 2^N-1 и преобразувате всяко от тези числа в двоична бройна система. Например за N=3 имате следните двоични представяния на числата 0 до 2^N-1:

   ```
   000, 001, 010, 011, 100, 101, 110, 111
   ```

   Сега за всяко двоично представяне взимате тези думи от множеството символни низове, за които имате единица на съответната позиция в двоичното представяне. Примерно за двоичното представяне "101" взимате първия и последния низ (там има единици) и пропускате втория низ (там има нула). Хитро, нали?

6. Ако се затрудните, потърсете "merge sort" в Интернет. Ще намерите стотици имплементации, включително на C#. Предизвикателството е да не се заделя при всяко рекурсивно извикване нов масив за резултата, защото това е неефективно, а да се ползват само 3 масива в цялата програма: двата масива, които се сливат и трети за резултата от сливането. Ще трябва да реализирате сливане две области от масив в област от друг масив.

7. Да предположим, че методът `Perm(k)` пермутира по всички възможни начини елементите от масив `p[]`, стоящи на позиции от 0 до k включително. В масива `p` първоначално записваме числата от 1 до N. Можем да реализираме рекурсивно `Perm(k)` по следния начин:

   1. При `k`=0 отпечатваме поредната пермутация и излизаме (дъно на рекурсията).
   2. За всяка позиция `i` от 0 до k-1 извършваме следното:
      a. Разменяме `p[i]` с `p[k]`.
      b. Извикваме рекурсия: `Perm(k-1)`.
      c. Разменяме обратно `p[i]` с `p[k]`.
   3. Извикваме `Perm(k-1)`.

   В началото започваме с извикване на `Perm(N-1)`.

8. Задачата не се различава съществено от задачата за намиране на всички подмножества измежду даден списък със символни низове.

Помислете ще работи ли бързо програмата при 500 числа? Обърнете внимание, че трябва да отпечатаме всички подмножества със сума N, които могат да бъдат ужасно много при голямо N и подходящи числа в масива. По тази причина задачата няма ефективно решение.

9. Ако подходите към проблема по метода на изчерпването на всички възможности, решението няма да работи при повече от 20-30 елемента. Затова може да подходите по съвсем различен начин в случай, че числата в масива са само положителни или са ограничени в някакъв диапазон (примерно `[-50…50]`). Тогава може да се използва следният оптимизационен алгоритъм с динамично оптимиране:

   Нека имаме масива с числа **`p[]`**. Нека означим с **`possible(k, sum)`** дали можем да получим сума **`sum`** като използваме само числата **`p[0]`**, **`p[1]`**, ..., **`p[k]`**. Тогава са в сила следните рекурентни зависимости:

   - **`possible(0, sum) = true`**, точно когато **`p[0] == sum`**
   - **`possible(k, sum) = true`**, точно когато **`possible[k-1, sum] == true`** или **`possible[k-1, sum-p[k]] == true`**

   Горната формула показва, че можем да получим сума **`sum`** от елементите на масива на позиции от `0` до **`k`**, ако едно от двете е в сила:

   - Елементът **`p[k]`** не участва в сумата **`sum`** и тя се получава по някакъв начин от останалите елементи (от `0` до **`k-1`**);
   - Елементът **`p[k]`** участва в сумата **`sum`**, а остатъкът **`sum-p[k]`** се получава по някакъв начин от останалите елементи (от `0` до **`k-1`**).

   Реализацията не е сложна, но трябва да внимавате и да не позволявате вече сметната стойност от двумерния масив **`possible[,]`** да се пресмята повторно. За целта трябва да пазите за всяко възможно **`k`** и **`sum`** стойността **`possible[k, sum]`**. Иначе алгоритъмът няма да работи при повече 20-30 елемента.

   Възстановяването на самите числа, които съставят намерената сума, може да се извърши като се тръгне отзад напред от сумата n, получена от първите k числа, като на всяка стъпка се търси как тази сума може да се получи чрез първите **`k-1`** числа (чрез взимане на **`k`**-тото число или пропускането му).

   Имайте предвид, че в общия случай всички възможни суми на числа от входния масив може да са ужасно много. Примерно всички възможни суми от 50 **`int`** числа в интервала [**`Int32.MinValue`** … **`Int32.MaxValue`**] са достатъчно много, че да не могат да се съберат в каквато и да е структура от данни. Ако обаче всички числа във входния масив са положителни (както е в нашата задача), може да пазите само сумите в интервала [1..N], защото от останалите са безперспективни и от тях не може да се получи търсената сума N чрез добавяне на едно или повече числа от входния масив.

Ако числата във входния масив не са задължително положителни, но са ограничени в някакъв интервал, тогава и всички възможни суми са ограничени в някакъв интервал и можем да ползваме описания по-горе алгоритъм. Например, ако диапазонът на числата във входния масив е от -50 до 50, то най-малката възможна сума е -50*N, а най-голямата е 50*N.

Ако числата във входния масив са произволни и не са ограничени в някакъв интервал, задачата няма ефективно решение.

Можете да прочетете повече за тази класическа оптимизационна задача в Уикипедия: http://en.wikipedia.org/wiki/Subset_sum_problem.

10. Прочетете за All Depth-First Search в интернет.
11. Потърсете в интернет за Depth-First Search или Breath-First Search.
12. Потърсете в интернет за Depth-First Search или Breath-First Search.
13. Помислете за подходяща реализация на алгоритъма за търсене в широчина (BFS). След като намерите площ, която отговаря на условията, направете всички нейни клетки непроходими, намерете следващата проходима клетка и потърсете от нея с BFS следващата площ от съседни проходими клетки.
14. Прочетете статията в Уикипедия: http://en.wikipedia.org/wiki/Breadth-first_search. Там има достатъчно обяснения за BFS и примерен код. За да реализирате опашка в C# използвайте обикновен масив или класа **`System.Collections.Generics.Queue<T>`**. За елементи в опашката използвайте собствена структура **`Point`** съдържаща x и y координати или кодирайте координатите в число или пък използвайте две опашки – по една за всяка от координатите.
15. За всяка папка (започвайки от `C:\`) принтирайте името и файловете на текущата директория и викайте рекурсивно своя метод за всяка поддиректория на текущата.

Telerik Academy

SCHOOL

Стани част от Академията на Телерик за ученици!

"Основната цел на обучението е да израстнеш, а нашите умове, за разлика от телата ни, могат да растат през целия ни живот!" Мортимър Адлър

АКАДЕМИЯТА НА ТЕЛЕРИК за ученици предоставя съвременно, специализирано обучение по софтуерно инженерство за ученици от средните училища в България. Нашата основна цел е да помогнем на младите момчета и момичета да се подготвят за Националната олимпиада по информационни технологии (НОИТ) и да придобият солидни теоретични и практически умения за разработка на софтуер и софтуерни технологии.

В Училищната академия на Телерик по софтуерно инженерство:

- **Стимулираме креативността и мисленето на учениците**
- **Всеки ученик получава индивидуално внимание и съдействие**
- **Полагаме основите за писане на качествен програмен код**
- **Изучаваме съвременните софтуерни технологии**
- **Подготвяме учениците за Националната олимпиада по информационни технологии**
- **Предлагаме голямо разнообразие от практически и теоретични обучения**

Програмата на Академията включва:

C#, .NET технологии, бази данни, SQL, Silverlight, 3D графики и моделиране, разработка на игри, програмиране на микроконтролери, WPF, ASP.NET, HTML5, разработка на мобилни приложения, основи на софтуерното инженерство, работа в екип

schoolacademy.telerik.com

groups.google.com/group/it-olymp *facebook.com/TelerikSchoolAcademy*

telerik
deliver more than expected

Глава 11. Създаване и използване на обекти

В тази тема...

В настоящата тема ще се запознаем накратко с основните понятия в обектно-ориентираното програмиране – класовете и обектите – и ще обясним как да използваме класовете от стандартните библиотеки на .NET Framework. Ще се спрем на някои често използвани системни класове и ще видим как се създават и използват техни инстанции (обекти). Ще разгледаме как можем да осъществяваме достъп до полетата на даден обект, как да извикваме конструктори и как да работим със статичните полета в класовете. Накрая ще се запознаем с понятието "пространства от имена" – с какво ни помагат, как да ги включваме и използваме.

Класове и обекти

През последните няколко десетилетия програмирането и информатиката като цяло претърпяват невероятно развитие и се появяват концепции, които променят изцяло начина, по който се изграждат програми. Точно такава радикална идея въвежда **обектно-ориентираното програмиране (ООП)**. Ще изложим кратко въведение в принципите на ООП и понятията, които се използват в него. Като начало ще обясним какво представляват класовете и обектите. Тези две понятия стоят в основата на ООП и са неразделна част от ежедневието на почти всеки съвременен програмист.

Какво е обектно-ориентирано програмиране?

Обектно-ориентираното програмиране е модел на програмиране, който използва **обекти** и техните взаимодействия за изграждането на компютърни програми. По този начин се постига лесен за разбиране, опростен модел на предметната област, който дава възможност на програмиста интуитивно (чрез проста логика) да решава много от задачите, които възникват в реалния свят.

Засега няма да навлизаме в детайли за това какви са целите и предимствата на ООП, както и да обясняваме подробно принципите при изграждане на йерархии от класове и обекти. Ще вмъкнем само, че програмните техники на ООП често включват капсулация, модулност, полиморфизъм и наследяване. Тези техники са извън целите на настоящата тема, затова ще ги разгледаме по-късно в главата "Принципи на обектно-ориентираното програмиране". Сега ще се спрем на обектите като основно понятие в ООП.

Какво е обект?

Ще въведем понятието **обект** в контекста на ООП. Софтуерните обекти моделират обекти от реалния свят или абстрактни концепции (които също разглеждаме като обекти).

Примери за реални обекти са хора, коли, стоки, покупки и т.н. Абстрактните обекти са понятия в някоя предметна област, които се налага да моделираме и използваме в компютърна програма. Примери за абстрактни обекти са структурите от данни стек, опашка, списък и дърво. Те не са предмет на настоящата тема, но ще ги разгледаме в детайли в следващите теми.

В обектите от реалния свят (също и в абстрактните обекти) могат да се отделят следните две групи техни характеристики:

- Състояния (states) – това са характеристики на обекта, които по някакъв начин го определят и описват по принцип или в конкретен момент.

- Поведения (behaviors) – това са специфични характерни действия, които обектът може да извършва.

Нека за пример вземем обектът от реалния свят "куче". Състояния на кучето могат да бъдат "име", "цвят на козината" и "порода", а негови поведения – "лаене", "седене" и "ходене".

Обектите в ООП обединяват данни и средствата за тяхната обработка в едно цяло. Те съответстват на обектите от реалния свят и съдържат в себе си данни и действия:

- Член-данни (data members) – представляват променливи, вградени в обектите, които описват състоянията им.
- Методи (methods) – вече сме ги разглеждали в детайли. Те са инструментът за изграждане на поведението на обектите.

Какво е клас?

Класът дефинира абстрактните характеристики на даден обект. Той е план или шаблон, чрез който се описва природата на нещо (някакъв обект). Класовете са градивните елементи на ООП и са неразделно свързани с обектите. Нещо повече, всеки обект е представител на точно един клас.

Ще дадем пример за клас и обект, който е негов представител. Нека имаме клас `Dog` и обект `Lassie`, който е представител на класа `Dog` (казваме още обект от тип `Dog`). Класът `Dog` описва характеристиките на всички кучета, докато `Lassie` е конкретно куче.

Класовете предоставят модулност и структурност на обектно-ориентираните програми. Техните характеристики трябва да са смислени в общ контекст, така че да могат да бъдат разбрани и от хора, които са запознати с проблемната област, без да са програмисти. Например, не може класът `Dog` да има характеристика "RAM памет" поради простата причина, че в контекста на този клас такава характеристика няма смисъл.

Класове, атрибути и поведение

Класът дефинира характеристиките на даден обект (които ще наричаме **атрибути**) и неговото **поведение** (действията, които обектът може да извършва). Атрибутите на класа се дефинират като собствени променливи в тялото му (наречени член-променливи). Поведението на обектите се моделира чрез дефиниция на методи в класовете.

Ще илюстрираме казаното дотук като дадем пример за реална дефиниция на клас. Нека се върнем отново на примера с кучето, който вече дадохме по-горе. Искаме да дефинираме клас `Dog`, който моделира реалният обект "куче". Класът ще включва характеристики, общи за всички кучета (като порода и цвят на козината), а също и характерно за кучетата поведение (като лаене, седене, ходене). В такъв случай ще имаме атрибути **`breed`** и `furColor`, а поведението ще бъде имплементирано чрез методите `Bark()`, `Sit()` и `Walk()`.

Обектите – инстанции на класовете

От казаното дотук знаем, че всеки обект е представител на точно един клас и е създаден по шаблона на този клас. Създаването на обект от вече дефиниран клас наричаме **инстанциране** (instantiation). **Инстанция** (instance) е фактическият обект, който се създава от класа по време на изпълнение на програмата.

Всеки обект е **инстанция** на конкретен клас. Тази инстанция се характеризира със **състояние** (state) – множество от стойности, асоциирани с атрибутите на класа.

В контекста на така въведените понятия, обектът се състои от две неща: моментното състояние и поведението, дефинирано в класа на обекта. Състоянието е специфично за инстанцията (обекта), но поведението е общо за всички обекти, които са представители на този клас.

Класове в C#

До момента разгледахме някои общи характеристики на ООП. Голяма част от съвременните езици за програмиране са обектно-ориентирани. Всеки от тях има известни особености при работата с класове и обекти. В настоящата книга ще се спрем само на един от тези езици – C#. Хубаво е да знаем, че знанията за ООП в C# ще бъдат от полза на читателя без значение кой обектно-ориентиран език използва в практиката, тъй като ООП е фундаментална концепция в програмирането, използвана от почти всички съвременни езици за програмиране.

Какво представляват класовете в C#?

Класът в C# се дефинира чрез ключовата дума `class`, последвана от идентификатор (име) на класа и съвкупност от член-данни и методи, обособени в собствен блок код.

Класовете в C# могат да съдържат следните елементи:

- Полета (fields) – член-променливи от определен тип;
- Свойства (properties) – това са специален вид елементи, които разширяват функционалността на полетата като дават възможност за допълнителна обработка на данните при извличането и записването им в полетата от класа. Ще се спрем по-подробно на тях в темата "Дефиниране на класове";
- Методи – реализират манипулацията на данните.

Примерен клас

Ще дадем пример за прост клас в C#, който съдържа изброените елементи. Класът **`Cat`** моделира реалния обект "котка" и притежава свойствата име и цвят. Посоченият клас дефинира няколко полета, свойства и методи, които по-късно ще използваме наготово. Следва дефиницията на класа (засега няма да разглеждаме в детайли дефиницията на класовете – ще обърнем специално внимание на това в главата "Дефиниране на класове"):

```
public class Cat
{
   // Field name
   private string name;
   // Field color
   private string color;

   public string Name
   {
      // Getter of the property "Name"
      get
      {
         return this.name;
      }
      // Setter of the property "Name"
      set
      {
         this.name = value;
      }
   }

   public string Color
   {
      // Getter of the property "Color"
      get
      {
         return this.color;
      }
      // Setter of the property "Color"
      set
      {
         this.color = value;
      }
   }

   // Default constructor
   public Cat()
   {
      this.name = "Unnamed";
      this.color = "gray";
   }

   // Constructor with parameters
   public Cat(string name, string color)
   {
      this.name = name;
      this.color = color;
   }

   // Method SayMiau
   public void SayMiau()
   {
      Console.WriteLine("Cat {0} said: Miauuuuuu!", name);
   }
}
```

Примерният клас **`Cat`** дефинира свойствата **`Name`** и **`Color`**, които пазят стойността си в скритите (private) полетата **`name`** и **`color`**. Допълнително са дефинирани два конструктора за създаване на инстанции от класа **`Cat`** съответно без и със параметри и метод на класа **`SayMiau()`**.

След като примерният клас е дефиниран, можем вече да го използваме, например по следния начин:

```
static void Main()
{
   Cat firstCat = new Cat();
   firstCat.Name = "Tony";
   firstCat.SayMiau();

   Cat secondCat = new Cat("Pepy", "Red");
   secondCat.SayMiau();
   Console.WriteLine("Cat {0} is {1}.",
      secondCat.Name, secondCat.Color);
}
```

Ако изпълним примера, ще получим следния резултат:

```
Cat Tony said: Miauuuuuu!
Cat Pepy said: Miauuuuuu!
Cat Pepy is Red.
```

Видяхме прост пример за дефиниране и използване на класове, а в секцията "Създаване и използване на обекти" ще обясним в подробности как се създават обекти, как се достъпват свойствата им и как се извикват методите им и това ще ни позволи да разберем как точно работи примерът.

Системни класове

Извикването на метода `Console.WriteLine(…)` на класа `System.Console` е пример за употребата на **системен клас** в C#. Системни наричаме класовете, дефинирани в стандартните библиотеки за изграждане на приложения със C# (или друг език за програмиране). Те могат да се използват във всички наши .NET приложения (в частност тези, които са написани на C#). Такива са например класовете `String`, `Environment` и `Math`, които ще разгледаме малко по-късно.

Важно е да се знае, че имплементацията на логиката в класовете е **капсулирана** (скрита) вътре в тях. За програмиста е от значение какво правят методите, а не как го правят и за това голяма част от класовете не е публично достъпна (`public`). При системните класове имплементацията често пъти дори изобщо не е достъпна за програмиста. По този начин се създават **нива на абстракция**, което е един от основните принципи в ООП.

Ще обърнем специално внимание на системните класове малко по-късно. Сега е време да се запознаем със създаването и използването на обекти в програмите.

Създаване и използване на обекти

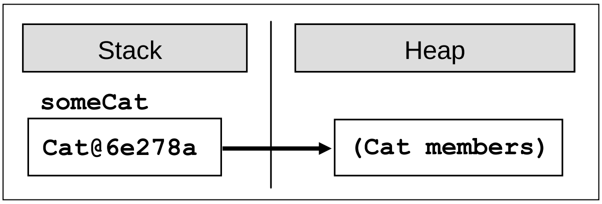
Засега ще се фокусираме върху създаването и използването на обекти в нашите програми. Ще работим с вече дефинирани класове и най-вече със системните класове от .NET Framework. Особеностите при дефинирането на наши собствени класове ще разгледаме по-късно в темата "Дефиниране на класове".

Създаване и освобождаване на обекти

Създаването на обекти от предварително дефинирани класове по време на изпълнението на програмата става чрез оператора `new`. Новосъздаденият обект обикновено се присвоява на променлива от тип, съвпадащ с класа на обекта (това, обаче, не е задължително - вижте глава "Принципи на обектно-ориентираното програмиране"). Ще отбележим, че при това присвояване същинският обект не се копира, а в променливата се записва само **референция** към новосъздадения обект (неговият адрес в паметта). Следва прост пример как става това:

```
Cat someCat = new Cat();
```

На променливата `someCat` от тип `Cat` присвояваме новосъздадена инстанция на класа `Cat`. Променливата `someCat` стои в стека, а нейната стойност (инстанцията на класа `Cat`) стои в динамичната памет (managed heap):



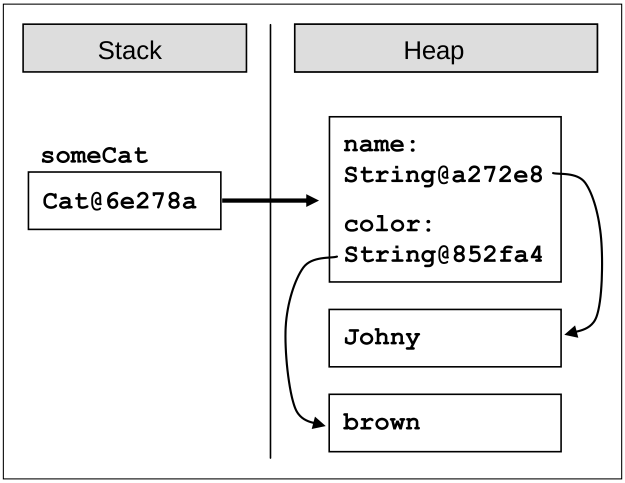
Създаване на обекти със задаване на параметри

Сега ще разгледаме леко променен вариант на горния пример, при който задаваме параметри при създаването на обекта:

```
Cat someCat = new Cat("Johnny", "brown");
```

В този случай искаме обектът **someCat** да представлява котка, която се казва "**Johnny**" и има кафяв цвят. Указваме това чрез думите "**Johnny**" и "**brown**", написани в скоби след името на класа.

При създаването на обект с оператора **new** се случват две неща: заделя се памет за този обект и се извършва начална инициализация на член-данните му. Инициализацията се осъществява от специален метод на класа, наречен **конструктор**. В горния пример инициализиращите параметри са всъщност параметри на конструктора на класа. Ще се спрем по-подробно на конструкторите след малко. Понеже член-променливите **name** и **color** на класа **Cat** са от референтен тип (от класа **String**), те се записват също в динамичната памет (heap) и в самия обект стоят техните референции (адреси). Следващата картинка показва това нагледно:

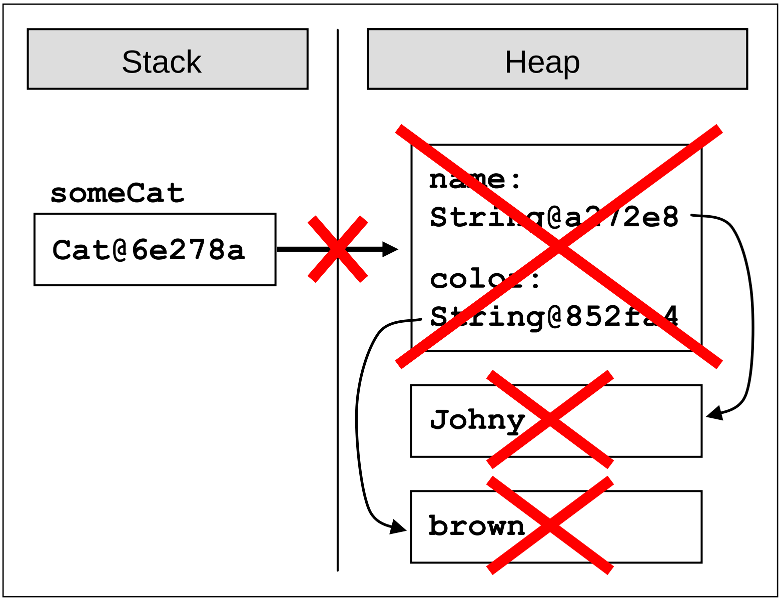


Освобождаване на обектите

Важна особеност на работата с обекти в C# е, че обикновено няма нужда от ръчното им разрушаване и освобождаване на паметта, заета от тях. Това е възможно поради вградената в .NET CLR система за почистване на паметта (**garbage collector**), която се грижи за освобождаването на неизползвани обекти вместо нас. Обектите, към които в даден момент вече няма референция в програмата, автоматично се унищожават и паметта, която заемат се освобождава. По този начин се предотвратяват много потенциални бъгове и проблеми. Ако искаме ръчно да освободим даден обект, трябва да унищожим референцията към него, например така:

```
someCat = null;
```

Това не унищожава обекта веднага, но го оставя в състояние, в което той е недостъпен от програмата и при следващото включване на системата за почистване на паметта (garbage collector), той ще бъде освободен:



Достъп до полета на обекта

Достъпът до полетата и свойствата (properties) на даден обект става чрез оператора `.` (точка), поставен между името на обекта и името на полето (или свойството). Операторът `.` не е необходим в случай, че достъпваме поле или свойство на даден клас в тялото на метод от същия клас.

Можем да достъпваме полетата и свойствата или с цел да извлечем данните от тях, или с цел да запишем нови данни. В случай на свойство, достъпът се реализира по абсолютно същия начин както и при поле – C# ни предоставя тази възможност. Това се постига чрез двете специални ключови думи **`get`** и **`set`** в дефиницията на свойството, които извършват съответно извличането на стойността на свойството и присвояването на нова стойност. В дефиницията на класа **`Cat`** (която дадохме по-горе) свойства са **`Name`** и **`Color`**.

Достъп до полета и свойства на обект – пример

Ще дадем прост пример за употребата на свойство на обект, като използваме вече дефинирания по-горе клас **`Cat`**. Създаваме инстанция **`myCat`** на класа **`Cat`** и присвояваме стойност **`"Alfred"`** на свойството **`Name`**. След това извеждаме на стандартния изход форматиран низ с името на нашата котка. Следва реализацията на примера:

```
class CatManipulating
{
   static void Main()
   {
      Cat myCat = new Cat();
      myCat.Name = "Alfred";

      Console.WriteLine("The name of my cat is {0}.",
         myCat.Name);
   }
}
```

Извикване на методи на обект

Извикването на методите на даден обект става отново чрез оператора **.** (точка). Операторът точка не е нужен единствено, в случай че съответният метод се извиква в тялото на друг метод от същия клас.

Сега е моментът да споменем факта, че методите на класовете имат **модификатори за достъп** **`public`**, **`private`** или **`protected`**, чрез които възможността за извикването им може да се ограничава. Ще разгледаме подробно тези модификатори в темата "Дефиниране на класове". Засега е достатъчно да знаем само, че модификаторът за достъп **`public`** не въвежда никакво ограничение за извикването на съответния метод, т.е. го прави публично достъпен.

Извикване на методи на обект – пример

Ще допълним примера, който вече дадохме като извикаме метода **`SayMiau`** на класа **`Cat`**. Ето какво се получава:

```
class CatManipulating
{
   static void Main()
   {
      Cat myCat = new Cat();
      myCat.Name = "Alfred";

      Console.WriteLine("The name of my cat is {0}.",
         myCat.Name);
      myCat.SayMiau();
   }
}
```

След изпълнението на горната програма на стандартния изход ще бъде изведен следният текст:

```
The name of my cat is Alfred.
Cat Alfred said: Miauuuuuu!
```

Конструктори

Конструкторът е специален метод на класа, който се извиква автоматично при създаването на обект от този клас и извършва инициализация на данните му (това е неговото основно предназначение). Конструкторът няма тип на връщана стойност и неговото име не е произволно, а задължително съвпада с името на класа. Конструкторът може да бъде със или без параметри.

Конструктор без параметри наричаме още конструктор по подразбиране (default constructor). Езикът C# допуска да няма изрично дефиниран конструктор в класа. В този случай, компилаторът създава автоматично празен конструктор по подразбиране, който занулява всички полета на класа.

Конструктори с параметри

Конструкторът може да приема параметри, както всеки друг метод. Всеки клас може да има произволен брой конструктори с единственото ограничение, че броят и типът на параметрите им трябва да бъдат различни. При създаването на обект от този клас се извиква точно един от дефинираните конструктори.

При наличието на няколко конструктора в един клас естествено възниква въпросът кой от тях се извиква при създаването на обект. Този проблем се решава по много интуитивен начин, както при методите. Подходящият конструктор се избира автоматично от компилатора в зависимост от подадената съвкупност от параметри при създаването на обекта. Използва се принципът на най-добро съвпадение.

Извикване на конструктори – пример

Да разгледаме отново дефиницията на класа **Cat** и по-конкретно двата конструктора на класа:

```
public class Cat
{
   // Field name
   private string name;
   // Field color
   private string color;
   …

   // Default constructor
   public Cat()
   {
      this.name = "Unnamed";
      this.color = "gray";
   }

   // Constructor with parameters
   public Cat(string name, string color)
   {
      this.name = name;
      this.color = color;
   }

   …
}
```

Ще използваме тези конструктори, за да илюстрираме употребата на конструктор без и с параметри. При така дефинирания клас **Cat** ще дадем пример за създаването на негови инстанции чрез всеки от двата конструктора. Единият обект ще бъде обикновена неопределена котка, а другият – нашата кафява котка Johnny. След това ще изпълним метода **SayMiau** на всяка от двете и ще разгледаме резултата. Следва изходният код:

```
class CatManipulating
{
   static void Main()
   {
      Cat someCat = new Cat();

      someCat.SayMiau();
      Console.WriteLine("The color of cat {0} is {1}.",
         someCat.Name, someCat.Color);

      Cat someCat = new Cat("Johnny", "brown");

      someCat.SayMiau();
      Console.WriteLine("The color of cat {0} is {1}.",
         someCat.Name, someCat.Color);
   }
}
```

В резултат от изпълнението на програмата се извежда следният текст на стандартния изход:

```
Cat Unnamed said: Miauuuuuu!
The color of cat Unnamed is gray.
```

```
Cat Johnny said: Miauuuuuu!
The color of cat Johnny is brown.
```

Статични полета и методи

Член-данните, които разглеждахме досега, реализират състояния на обектите и са пряко свързани с конкретни инстанции на класовете. В ООП има специална категория полета и методи, които се асоциират с тип данни (клас), а не с конкретна негова инстанция (обект). Наричаме ги **статични членове** (static members), защото са независими от конкретните обекти. Нещо повече, те се използват без да има създадена инстанция на класа, в който са дефинирани. Те могат да бъдат полета, методи и конструктори. Да разгледаме накратко статичните членове в C#.

Статично поле или метод в даден клас се дефинира чрез ключовата дума **`static`**, поставена преди типа на полето или типа на връщаната стойност на метода. При дефинирането на статичен конструктор думата **`static`** се поставя преди името на конструктора. Статичните конструктори не са предмет на настоящата тема – засега ще се спрем само на статичните полета и методи (по-любознателните читатели могат да направят справка в MSDN).

Кога да използваме статични полета и методи?

За да отговорим на този въпрос трябва преди всичко добре да разбираме разликата между статичните и нестатичните (non-static) членове. Ще разгледаме по-детайлно каква е тя.

Вече обяснихме основната разлика между двата вида членове. Нека интерпретираме класа като категория обекти, а обектът – като елемент, представител на тази категория. Тогава статичните членове отразяват състояния и поведения на самата категория обекти, а нестатичните – състояния и поведения на отделните представители на категорията.

Сега ще обърнем по-специално внимание на инициализацията на статичните и нестатичните полета. Вече знаем, че нестатичните полета се инициализират заедно с извикването на конструктор на класа при създаването на негова инстанция – или в тялото на конструктора, или извън него. Инициализацията на статичните полета, обаче, не може да става при създаването на обект от класа, защото те могат да бъдат използвани, без да има създадена инстанция на този клас. Важно е да се знае следното:

Статичните полета се инициализират, когато типът данни (класът) се използва за пръв път по време на изпълнението на програмата.

Време е да видим как се използват статични полета и методи на практика.

Статични полета и методи – пример

Примерът, който ще дадем решава следната проста задача: нужен ни е метод, който всеки път връща стойност с едно по-голяма от стойността, върната при предишното извикване на метода. Избираме първата върната от метода стойност да бъде 0. Очевидно такъв метод генерира редицата на естествените числа. Подобна функционалност има широко приложение в практиката, например за унифицирано номериране на обекти. Сега ще видим как може да се реализира с инструментите на ООП.

Да приемем, че методът е наречен **`NextValue()`** и е дефиниран в клас с име **`Sequence`**. Класът има поле **`currentValue`** от тип **`int`**, което съдържа последно върнатата стойност от метода. Искаме в тялото на метода да се извършват последователно следните две действия: да се увеличава стойността на полето и да се връща като резултат новата му стойност. Връщаната от метода стойност очевидно не зависи от конкретна инстанция на класа **`Sequence`**. Поради тази причина методът и полето са статични. Следва описаната реализация на класа:

```
public class Sequence
{
   // Static field, holding the current sequence value
   private static int currentValue = 0;

   // Intentionally deny instantiation of this class
   private Sequence()
   {
   }

   // Static method for taking the next sequence value
   public static int NextValue()
   {
      currentValue++;
      return currentValue;
   }
}
```

Наблюдателният читател е забелязал, че така дефинираният клас има конструктор по подразбиране, който е деклариран като **`private`**. Тази употреба на конструктор може да изглежда особена, но е съвсем умишлена. Добре е да знаем следното:

Клас, който има само `private` конструктори не може да бъде инстанциран. Такъв клас обикновено има само статични членове и се нарича utility клас.

Засега няма да навлизаме в детайли за употребата на **модификаторите за достъп `public`, `private` и `protected`**. Ще ги разгледаме подробно в главата "Дефиниране на класове".

Нека сега видим една проста програма, която използва класа **`Sequence`**:

```
class SequenceManipulating
{
   static void Main()
   {
      Console.WriteLine("Sequence[1..3]: {0}, {1}, {2}",
         Sequence.NextValue(), Sequence.NextValue(),
         Sequence.NextValue());
   }
}
```

Примерът извежда на стандартния изход първите три естествени числа чрез трикратно последователно извикване на метода **`NextValue()`** от класа **`Sequence`**. Резултатът от този код е следният:

```
Sequence[1..3]: 1, 2, 3
```

Ако се опитаме да създадем няколко различни редици, понеже конструкторът на класа **`Sequence`** е деклариран като **`private`**, ще получим грешка по време на компилация.

Примери за системни C# класове

След като вече се запознахме с основната функционалност на обектите, ще разгледаме накратко няколко често използвани системни класа от стандартните библиотеки на .NET Framework. По този начин ще видим на практика обясненото до момента, а също ще покажем как системните класове улесняват ежедневната ни работа.

Класът System.Environment

Започваме с един от основните системни класове в .NET Framework. Той съдържа набор от полезни полета и методи, които улесняват получаването на информация за хардуера и операционната система, а някои от тях дават възможност за взаимодействие с обкръжението на програмата. Ето част от функционалността, която предоставя този клас:

- Информация за броя на процесорите, мрежовото име на компютъра, версията на операционната система, името на текущия потребител, текущата директория и др.
- Достъп до външно дефинирани свойства (properties) и променливи на обкръжението (environment variables), които няма да разглеждаме в настоящата книга.

Сега ще покажем едно интересно приложение на метод от класа **`Environment`**, което често се използва в практиката при разработката на програми с критично бързодействие. Ще засечем времето за изпълнение на фрагмент от изходния код с помощта на свойството **`TickCount`**. Ето как може да стане това:

```
class SystemTest
{
   static void Main()
   {
      int sum = 0;
      int startTime = Environment.TickCount;

      // The code fragment to be tested
      for (int i = 0; i < 10000000; i++)
      {
         sum++;
      }

      int endTime = Environment.TickCount;
      Console.WriteLine("The time elapsed is {0} sec.",
         (endTime - startTime) / 1000.0);
   }
}
```

Статичното свойство **`TickCount`** от класа **`Environment`** връща като резултат броя милисекунди, които са изтекли от включването на компютъра до момента на извикването на метода. С негова помощ засичаме изтеклите милисекунди преди и след изпълнението на критичния код. Тяхната разлика е всъщност търсеното време за изпълнение на фрагмента код, измерено в милисекунди.

В резултат от изпълнението на програмата на стандартния изход се извежда резултат от следния вид (засеченото време варира в зависимост от конкретната компютърна конфигурация и нейното натоварване):

```
The time elapsed is 0,031 sec.
```

В примера използвахме два статични члена от два системни класа: статичното свойство **`Environment.TickCount`** и статичния метод **`Console.WriteLine(…)`**.

Класът System.String

Вече сме споменавали класа **`string`** (**`System.String`**) от .NET Framework, който представя символни низове (последователности от символи). Да припомним, че можем да считаме низовете за примитивен тип данни в C#, въпреки че работата с тях се различава до известна степен от работата с другите примитивни типове (цели и реални числа, булеви променливи и др.). Ще се спрем по-подробно на тях в темата "Символни низове".

Класът System.Math

Класът **`System.Math`** съдържа методи за извършването на основни числови операции като повдигане в степен, логаритмуване, коренуване и някои тригонометрични функции. Ще дадем един прост пример, който илюстрира употребата му.

Искаме да съставим програма, която пресмята лицето на триъгълник по дадени дължини на две от страните и ъгъла между тях в градуси. За тази цел имаме нужда от метода **`Sin(…)`** и константата **`PI`** на класа **`Math`**. С помощта на числото π лесно преобразуваме към радиани въведеният в градуси ъгъл. Следва примерна реализация на описаната логика:

```
class MathTest
{
  static void Main()
  {
    Console.WriteLine("Length of the first side:");
    double a = double.Parse(Console.ReadLine());
    Console.WriteLine("Length of the second side:");
    double b = double.Parse(Console.ReadLine());
    Console.WriteLine("Size of the angle in degrees:");
    int angle = int.Parse(Console.ReadLine());;

    double angleInRadians = Math.PI * angle / 180.0;
    Console.WriteLine("Face of the triangle: {0}",
      0.5 * a * b * Math.Sin(angleInRadians));
  }
}
```

Можем лесно да тестваме програмата като проверим дали пресмята правилно лицето на равностранен триъгълник. За допълнително улеснение избираме дължина на страната да бъде 2 – тогава лицето му намираме с добре известната формула:

$$S = \frac{\sqrt{3}}{4}2^2 = \sqrt{3} = 1{,}7320508...$$

Въвеждаме последователно числата 2, 2, 60 и на стандартния изход се извежда:

```
Face of the triangle: 1,73205080756888
```

Класът System.Math – още примери

Както вече видяхме, освен математически методи, класът **`Math`** дефинира и две добре известни в математиката константи: тригонометричната константа π и Неперовото число ***e***. Ето още един пример за тях:

```
Console.WriteLine(Math.PI);
Console.WriteLine(Math.E);
```

При изпълнение на горния код се получава следния резултат:

```
3.141592653589793
2.718281828459045
```

Класът System.Random

Понякога в програмирането се налага да използваме случайни числа. Например искаме да генерираме 6 случайни числа в интервала между 1 и 49 (не непременно различни). Това можем да направим използвайки класа **`System.Random`** и неговия метод **`Next()`**. Преди да използваме класа **`Random`** трябва да създадем негова инстанция, при което тя се инициализира със случайна стойност (извлечена от текущото системно време в операционната система). След това можем да генерираме случайно число в интервала **`[0…n)`** чрез извикване на метода **`Next(n)`**. Забележете, че този метод може да върне нула, но връща винаги случайно число по-малко от зададената стойност **`n`**. Затова, ако искаме да получим число в интервала **`[1…49]`**, трябва да използваме израза **`Next(49) + 1`**. Следва примерен изходен код на програма, която, използвайки класа **`Random`**, генерира 6 случайни числа в интервала от 1 до 49:

```
class RandomNumbersBetween1and49
{
   static void Main()
   {
      Random rand = new Random();
      for (int number = 1; number <= 6; number++)
      {
         int randomNumber = rand.Next(49) + 1;
         Console.Write("{0} ", randomNumber);
      }
   }
}
```

Ето как изглежда един възможен изход от работата на програмата:

```
16 49 7 29 1 28
```

Класът Random – още един пример

За да ви покажем колко полезен може да е генераторът на случайни числа в .NET Framework, ще си поставим за задача да генерираме случайна парола, която е дълга между 8 и 15 символа, съдържа поне две главни букви, поне две малки букви, поне една цифра и поне три специални знака. За целта ще използваме следния алгоритъм:

1. Започваме от празна парола. Създаваме генератор на случайни числа.
2. Генерираме два пъти по една случайна главна буква и я поставяме на случайна позиция в паролата.

3. Генерираме два пъти по една случайна малка буква и я поставяме на случайна позиция в паролата.
4. Генерираме една случайна цифра и я поставяме на случайна позиция в паролата.
5. Генерираме три пъти по един случаен специален символ и го поставяме на случайна позиция в паролата.
6. До момента паролата трябва да се състои от 8 знака. За да я допълним до най-много 15 символа, можем случаен брой пъти (между 0 и 7) да вмъкнем на случайна позиция в паролата случаен знак (главна буква или малка буква или цифра или специален символ).

Следва имплементация на описания алгоритъм:

```
class RandomPasswordGenerator
{
  private const string CapitalLetters =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  private const string SmallLetters =
    "abcdefghijklmnopqrstuvwxyz";
  private const string Digits = "0123456789";
  private const string SpecialChars =
    "~!@#$%^&*()_+=`{}[]\\|':;.,/?<>";
  private const string AllChars =
    CapitalLetters + SmallLetters + Digits + SpecialChars;

  private static Random rnd = new Random();

  static void Main()
  {
    StringBuilder password = new StringBuilder();

    // Generate two random capital letters
    for (int i = 1; i <= 2; i++)
    {
      char capitalLetter = GenerateChar(CapitalLetters);
      InsertAtRandomPosition(password, capitalLetter);
    }

    // Generate two random small letters
    for (int i = 1; i <= 2; i++)
    {
      char smallLetter = GenerateChar(SmallLetters);
      InsertAtRandomPosition(password, smallLetter);
    }

    // Generate one random digit
    char digit = GenerateChar(Digits);
    InsertAtRandomPosition(password, digit);
    // Generate 3 special characters
    for (int i = 1; i <= 3; i++)
    {
      char specialChar = GenerateChar(SpecialChars);
      InsertAtRandomPosition(password, specialChar);
    }

    // Generate few random characters (between 0 and 7)
    int count = rnd.Next(8);
    for (int i = 1; i <= count; i++)
    {
      char specialChar = GenerateChar(AllChars);
      InsertAtRandomPosition(password, specialChar);
    }

    Console.WriteLine(password);
  }

  private static void InsertAtRandomPosition(
    StringBuilder password, char character)
  {
    int randomPosition = rnd.Next(password.Length + 1);
    password.Insert(randomPosition, character);
  }

  private static char GenerateChar(string availableChars)
  {
    int randomIndex = rnd.Next(availableChars.Length);
    char randomChar = availableChars[randomIndex];
    return randomChar;
  }
}
```

Нека обясним някои неясни моменти в изходния код. Да започнем от дефинициите на константи:

```
private const string CapitalLetters =
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
private const string SmallLetters =
  "abcdefghijklmnopqrstuvwxyz";
private const string Digits = "0123456789";
private const string SpecialChars =
  "~!@#$%^&*()_+=`{}[]\\|':;.,/?<>";
private const string AllChars =
  CapitalLetters + SmallLetters + Digits + SpecialChars;
```

Константите в C# представляват неизменими променливи, чиито стойности се задават по време на инициализацията им в изходния код на програмата и след това не могат да бъдат променяни. Те се декларират с модификатора `const`. Използват се за дефиниране на дадено число или низ, което се използва след това многократно в програмата. По този начин се спестяват повторенията на определени стойности в кода и се позволява лесно тези стойности да се променят чрез промяна само на едно място в кода. Например ако в даден момент решим, че символът "," (запетая) не трябва да се ползва при генерирането на пароли, можем да променим само един ред в програмата (съответната константа) и промяната ще се отрази навсякъде, където е използвана константата. Константите в C# се изписват в Pascal Case (думите в името са залепени една за друга, като всяка от тях започва с главна буква, а останалите букви са малки).

Нека обясним и как работят останалите части от програмата. В началото като статична член-променлива в класа **`RandomPasswordGenerator`** се създава генераторът на случайни числа **`rnd`**. Понеже тази променлива **`rnd`** е дефинирана в самия клас (не в **`Main()`** метода), тя е достъпна от целия клас (от всички негови методи) и понеже е обявена за статична, тя е достъпна и от статичните методи. По този начин навсякъде, където програмата има нужда от случайна целочислена стойност, се използва един и същ генератор на случайни числа, който се инициализира при зареждането на класа **`RandomPasswordGenerator`**.

Методът **`GenerateChar()`** връща случайно избран символ измежду множество символи, подадени му като параметър. Той работи много просто: избира случайна позиция в множеството символи (между 0 и броя символи минус 1) и връща символът на тази позиция.

Методът **`InsertAtRandomPosition()`** също не е сложен. Той избира случайна позиция в **`StringBuilder`** обекта, който му е подаден и вмъква на тази позиция подадения символ. На класа **`StringBuilder`** ще обърнем специално внимание в главата "Символни низове".

Ето примерен изход от програмата за генериране на пароли, която разгледахме и обяснихме как работи:

```
8p#Rv*yTl{tN4
```

Пространства от имена

Пространство от имена (namespace / package) в ООП наричаме контейнер за група класове, които са обединени от общ признак или се използват в общ контекст. Пространствата от имена спомагат за една по-добра логическа организация на изходния код като създават семантично разделение на класовете в категории и улесняват употребата им в програмния код. Сега ще се спрем на пространствата в C# и ще видим как можем да ги използваме.

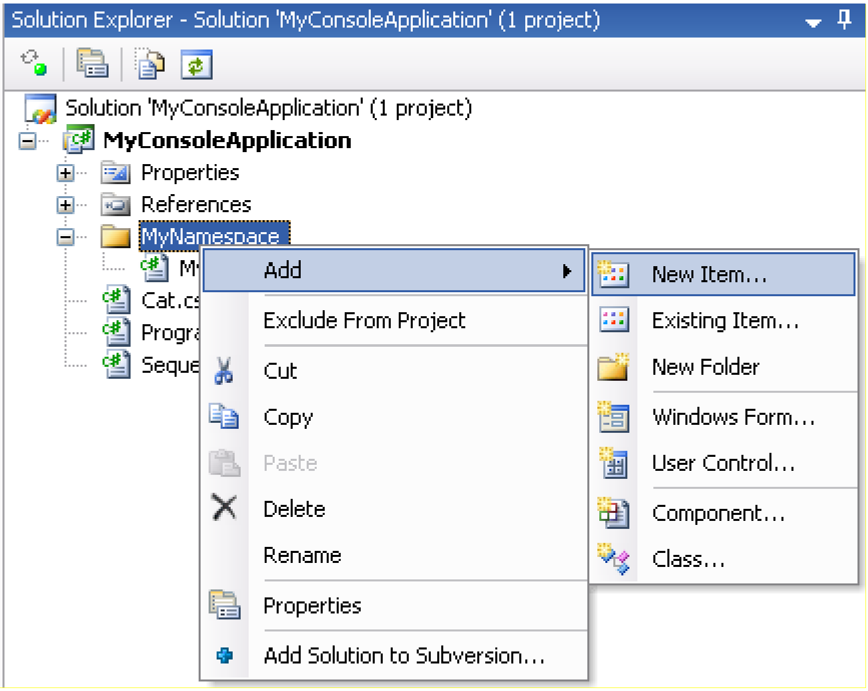
Какво представляват пространствата от имена в C#?

Пространствата от имена (namespaces) в C# представляват именувани групи класове, които са логически свързани, без да има специално изискване как да бъдат разположени във файловата система. Прието е, обаче, името на папката да съвпада с името на пространството и имената на файловете да съвпадат с имената на класовете, които се съхраняват в тях. Трябва да отбележим, че в някои езици за програмиране компилацията на изходния код на дадено пространство зависи от разпределението на елементите на пространството в папки и файлове на диска. В Java, например, така описаната файлова организация на пространствата е напълно задължителна (ако не е спазена, възниква грешка при компилацията). Езикът C# не е толкова стриктен в това отношение.

Нека сега разгледаме механизма за дефиниране на пространства.

Дефиниране на пространства от имена

В случай, че искаме да създадем ново пространство или да създадем нов клас, който ще принадлежи на дадено пространство, във Visual Studio това става автоматично чрез командите в контекстното меню на Solution Explorer (при щракане с десния бутон на мишката върху съответната папка). Solution Explorer по подразбиране се визуализира като страница в дясната част на интегрираната среда. Ще покажем нагледно как можем да добавим нов клас към вече съществуващото пространство `MyNamespace` чрез контекстното меню на Solution Explorer във Visual Studio:


Solution Explorer - Solution 'MyConsoleApplication' (1 project)
Solution 'MyConsoleApplication' (1 project)
MyConsoleApplication
Properties
References
MyNamespace
M
Cat.cs
Progr
Seque
Add
Exclude From Project
Cut
Copy
Paste
Delete
Rename
Properties
Add Solution to Subversion...
New Item...
Existing Item...
New Folder
Windows Form...
User Control...
Component...
Class...

Тъй като проектът ни се нарича **`MyConsoleApplication`** и добавяме нов клас в неговата подпапка **`MyNamespace`**, новосъздаденият клас ще бъде в следното пространство:

```
namespace MyConsoleApplication.MyNamespace
```

Ако сме дефинирали клас в собствен файл и искаме да го добавим към ново или вече съществуващо пространство, не е трудно да го направим ръчно. Достатъчно е да променим именувания блок с ключова дума **`namespace`**, в който се намира класа:

```
namespace <namespace_name>
{
   ...
}
```

При дефиницията използваме ключовата дума **`namespace`**, последвана от пълното име на пространството. Прието е имената на пространствата в C# да започват с главна буква и да бъдат изписвани в Pascal Case. Например, ако трябва да направим пространство, което съдържа класове за работа със символни низове, желателно е да го именуваме **`StringUtils`**, а не **`string_utils`**.

Вложени пространства

Освен класове, пространствата могат да съдържат в себе си и други пространства (вложени пространства, nested namespaces). По този начин съвсем интуитивно се изгражда йерархия от пространства, която позволява още по-прецизно разделение на класовете според тяхната семантика.

При назоваването на пространствата в йерархията се използва символът **`.`** за разделител (точкова нотация). Например пространството **`System`** от .NET Framework съдържа в себе си подпространството **`Collections`** и така пълното название на вложеното пространство **`Collections`** добива вида **`System.Collections`**.

Пълни имена на класовете

За да разберем напълно смисъла на пространствата, важно е да знаем следното:

Класовете трябва да имат уникални имена само в рамките на пространството от имена, в което са дефинирани.

Извън дадено пространство може да има класове с произволни имена, без значение дали съвпадат с някои от имената на класовете в пространството. Това е така, защото класовете в пространството са определени еднозначно от неговия контекст. Време е да видим как се определя синтактично тази еднозначност.

Пълно име на клас наричаме собственото име на класа, предшествано от името на пространството, в което този клас е дефиниран. Пълното име на всеки клас е уникално. Отново се използва точковата нотация:

```
<namespace_name>.<class_name>
```

Нека вземем за пример системния клас **`CultureInfo`**, дефиниран в пространството **`System.Globalization`** (вече сме го използвали в темата "Вход и изход от конзолата"). Съгласно дадената дефиниция, пълното име на този клас е **`System.Globalization.CultureInfo`**.

В .NET Framework понякога има класове от различни пространства със съвпадащи имена, например:

```
System.Windows.Forms.Control
System.Web.UI.Control
System.Windows.Controls.Control
```

Включване на пространство

При изграждането на приложения в зависимост от предметната област често се налага многократното използване на класове някое пространство. За удобство на програмиста има механизъм за **включване** на пространство към текущия файл със сорс код. След като е включено дадено пространство, всички класове дефинирани в него могат свободно да се използват, без да е необходимо използването на техните пълни имена.

Включването на пространство към файл с изходен код се извършва чрез ключовата дума **`using`** по следния начин:

```
using <namespace_name>;
```

Ще обърнем внимание на една важна особеност при включването на пространства по описания начин. Всички класове, които се съдържат директно в пространството **`<namespace_name>`** са включени и могат да се използват, но трябва да знаем следното:

Включването на пространства не е рекурсивно, т.е. при включване на пространство не се включват класовете от вложените в него пространства.

Например включването на пространството от имена **`System.Collections`** не включва автоматично класовете, съдържащи се в пространството от имена **`System.Collections.Generic`**. При употребата им трябва да ги назоваваме с пълните им имена или да включим изрично пространството, в което се намират.

Включване на пространство – пример

За да илюстрираме принципа на включването на пространство, ще разгледаме следната програма, въвежда списъци от числа и брои колко от тях са цели и колко от тях са дробни:

```
class NamespaceImportTest
{
   static void Main()
   {
      System.Collections.Generic.List<int> ints =
         new System.Collections.Generic.List<int>();
      System.Collections.Generic.List<double> doubles =
         new System.Collections.Generic.List<double>();

      while (true)
      {
         int intResult;
         double doubleResult;
         Console.WriteLine("Enter an int or a double:");
         string input = Console.ReadLine();

         if (int.TryParse(input, out intResult))
         {
            ints.Add(intResult);
         }
         else if (double.TryParse(input, out doubleResult))
         {
            doubles.Add(doubleResult);
         }
         else
         {
            break;
         }
      }

      Console.Write("You entered {0} ints:", ints.Count);
      foreach (var i in ints)
      {
         Console.Write(" " + i);
      }
      Console.WriteLine();

      Console.Write("You entered {0} doubles:", doubles.Count);
      foreach (var d in doubles)
      {
         Console.Write(" " + d);
      }
      Console.WriteLine();
   }
}
```

За целта програмата използва класа **System.Collections.Generic.List** като го назовава с пълното му име.

Нека сега видим как работи горната програма: въвеждаме последователно стойностите **4, 1.53, 0.26, 7, 2, end.** Получаваме следния резултат на стандартния изход:

```
You entered 3 ints: 4 7 2
You entered 2 doubles: 1.53 0.26
```

Програмата извършва следното: дава на потребителя възможност да въвежда последователно числа, които могат да бъдат цели или реални. Въвеждането продължава до момента, в който бъде въведена стойност, различна от число. След това на стандартния изход се извеждат два реда съответно с целите и с реалните числа.

За реализацията на описаните действия използваме два помощни обекта съответно от тип **System.Collections.Generic.List<int>** и **System.Collections.Generic.List<double>**. Очевидно е, че пълните имена на класовете правят кода непрегледен и труден за четене и създават неудобства. Можем лесно да избегнем този ефект като включим пространството **System.Collections.Generic** и използваме директно класовете по име. Следва промененият вариант на горната програма:

```
using System.Collections.Generic;

class NamespaceImportTest
{
    static void Main()
    {
        List<int> ints = new List<int>();
        List<double> doubles = new List<double>();
        …
    }
}
```

Упражнения

1. Напишете програма, която прочита от конзолата година и проверява дали е високосна.
2. Напишете програма, която генерира и принтира на конзолата 10 случайни числа в интервала [100, 200].
3. Напишете програма, която извежда на конзолата кой ден от седмицата е днес.
4. Напишете програма, която извежда на стандартния изход броя на дните, часовете и минутите, които са изтекли от включването на ком-

пютъра до момента на изпълнението на програмата. За реализацията използвайте класа **`Environment`**.

5. Напишете програма, която по дадени два катета намира хипотенузата на правоъгълен триъгълник. Реализирайте въвеждане на дължините на катетите от стандартния вход, а за пресмятането на хипотенузата използвайте методи на класа **`Math`**.

6. Напишете програма, която пресмята лице на триъгълник по:

   a. дължините на трите му страни;

   b. дължината на една от страните и височината към нея;

   c. дължините на две от страните му и ъгъла между тях в градуси.

7. Дефинирайте свое собствено пространство **`Chapter11`** и поставете в него двата класа **`Cat`** и **`Sequence`**, които използвахме в примерите на текущата тема. Направете още едно собствено пространство с име **`Chapter11.Examples`** и в него направете клас, който извиква класовете **`Cat`** и **`Sequence`**.

8. Напишете програма, която създава 10 обекта от тип **`Cat`**, дава им имена от вида **`CatN`**, където **`N`** е уникален пореден номер на обекта, и накрая извиква метода **`SayMiau()`** на всеки от тях. За реализацията използвайте вече дефинираното пространство **`Chapter11`**.

9. Напишете програма, която пресмята броя работни дни между днешната дата и дадена друга дата след днешната (включително). Работните дни са всички дни без събота и неделя, които не са официални празници, като по изключение събота може да е работен ден, когато се отработват почивни дни около празниците. Програмата трябва да пази списък от предварително зададени официални празници, както и списък от предварително зададени работни съботи.

10. Дадена е последователност от цели положителни числа, записани едно след друго като символен низ, разделени с интервал. Да се напише програма, която пресмята сумата им. Пример: "**`43 68 9 23 318`**" → **`461`**.

11. Напишете програма, която генерира случайно рекламно съобщение за някакъв продукт. Съобщенията трябва да се състоят от хвалебствена фраза, следвани от хвалебствена случка, следвани от автор (първо и второ име) и град, които се избират от предварително подготвени списъци. Например, нека имаме следните списъци:

    - Хвалебствени фрази: {"Продуктът е отличен.", "Това е страхотен продукт.", "Постоянно ползвам този продукт.", "Това е най-добрият продукт от тази категория."}.

    - Хвалебствени случки: {"Вече се чувствам добре.", "Успях да се променя.", "Той направи чудо.", "Не мога да повярвам, но вече се чувствам страхотно.", "Опитайте и вие. Аз съм много доволна."}.

- Първо име на автор: {"Диана", "Петя", "Стела", "Елена", "Катя"}.
- Второ име на автор: {"Иванова", "Петрова", "Кирова"}.
- Градове: {"София", "Пловдив", "Варна", "Русе", "Бургас"}.

Тогава програма би могла да изведе следното случайно-генерирано рекламно съобщение:

```
Постоянно ползвам този продукт. Опитайте и вие. Аз съм
доволна. -- Елена Петрова, Варна
```

12. * Напишете програма, която изчислява стойността на даден числен израз, зададен като стринг. Численият израз се състои от:
    - реални числа, например **`5`**, **`18.33`**, **`3.14159`**, **`12.6`**;
    - аритметични оператори: **`+`**, **`-`**, **`*`**, **`/`** (със стандартните им приоритети);
    - математически функции: **`ln(x)`**, **`sqrt(x)`**, **`pow(x,y)`**;
    - скоби за промяна на приоритета на операциите: **`(`** и **`)`**.

    Обърнете внимание, че числовите изрази имат приоритет, например изразът **`-1 + 2 + 3 * 4 - 0.5 = (-1) + 2 + (3 * 4) - 0.5 = 12.5`**.

Решения и упътвания

1. Използвайте структурата **`DateTime`**.
2. Използвайте класа **`Random`**. Можете да генерирате произволни числа в интервала [0, 100] и към всички да прибавите 100.
3. Използвайте структурата **`DateTime`**.
4. Използвайте свойството **`Environment.TickCount`**, за да получите броя на изтеклите милисекунди. Използвайте факта, че в една секунда има 1000 милисекунди и пресметнете минутите, часовете и дните.
5. Хипотенузата на правоъгълен триъгълник се намира с помощта на известната теорема на Питагор: $a^2 + b^2 = c^2$, където **`a`** и **`b`** са двата катета, а **`c`** е хипотенузата. Коренувайте двете страни, за да получите формула за дължината на хипотенузата. За реализацията на коренуването използвайте метода **`Sqrt(…)`** на класа **`Math`**.
6. За първата подточка на задачата използвайте Хероновата формула: $S = \sqrt{p(p-a)(p-b)(p-c)}$, където $p = \frac{a+b+c}{2}$. За втората подточка използвайте формулата: $S = \frac{a * h_a}{2}$. За третата използвайте

формулата: $S = \frac{a * b * sin(\gamma)}{2}$. За функцията синус използвайте класа `System.Math`.

7. Създайте нов проект във Visual Studio, щракнете с десния бутон върху папката му и изберете от контекстното меню Add -> New Folder. След като въведете име на папката и натиснете [Enter], щракнете с десния бутон върху новосъздадената папка и изберете Add -> New Item… От списъка изберете Class, за име на новия клас въведете `Cat` и натиснете [Add]. Подменете дефиницията на новосъздадения клас с дефиницията, която дадохме в тази тема. Направете същото за класа `Sequence`.

8. Създайте масив с 10 елемента от тип `Cat`. Създайте в цикъл 10 обекта от тип `Cat` (използвайте конструктор с параметри), като ги присвоявате на съответните елементи от масива. За поредния номер на обектите използвайте метода `NextValue()` на класа `Sequence`. Накрая отново в цикъл изпълнете метода `SayMiau()` на всеки от елементите на масива.

9. Използвайте класа `System.DateTime` и методите в него. Можете да завъртите цикъл от днешната дата (`DateTime.Now.Date`) до крайната дата, увеличавайки последователно деня чрез метода `AddDays(1)`.

10. Използвайте `String.Split(' ')`, за да разцепите символния низ по интервалите, след което с `Int32.Parse(…)` можете да извлечете отделните числа от получения масив от символни низове.

11. Използвайте класа `System.Random` и неговия метод `Next(…)`.

12. Задачата за пресмятане на числов израз е доста трудна и е малко вероятно да я решите коректно без да прочетете от някъде как се решава. За начало разгледайте статиите в Wikipedia за "Shunting-yard algorithm" (http://en.wikipedia.org/wiki/Shunting-yard_algorithm), която описва как се преобразува израз от нормален в обратен полски запис (postfix notation), и статията за пресмятане на постфиксен израз (http://en.wikipedia.org/wiki/Reverse_Polish_notation).

Telerik Academy

Присъедини се към Академията на Телерик!

АКАДЕМИЯТА НА ТЕЛЕРИК предоставя безплатно практическо обучение, насочено към всички млади хора, желаещи да станат умели .NET софтуерни инженери и да се присъединят към екипа на Телерик.

Академията подготвя бъдещите софтуерни специалисти за сложността, разнообразието и динамиката на днешната ИТ действителност, като за целта всички курсисти се обучават да работят с най-новите технологии на Майкрософт и черпят знания и опит от доказали се в областта на програмирането лектори и разработчици на софтуер.

В академията ще получите задълбочени знания и опит, изучавайки:

C#, .NET технологии, бази данни, SQL, Silverlight, WPF, ASP.NET, HTML5, разработка на мобилни приложения за iOS, Android и Windows Phone, основите на софтуерното инженерство

Академията на Телерик ви дава възможност да:

- **Учите напълно БЕЗПЛАТНО**
- **Изберете сред редица РАЗЛИЧНИ КУРСОВЕ**
- **Овладеете ОСНОВИТЕ на софтуерното инженерство**
- **Усвоите ПРОЦЕСА за разработка на софтуер**
- **Получите задълбочени теоретични и практически ИТ ПОЗНАНИЯ**
- **Станете умел .NET СОФТУЕРЕН ИНЖЕНЕР**
- **Започнете своята ИТ кариера в ТЕЛЕРИК – РАБОТОДАТЕЛ #1 в България за 2010 г.**

Само в рамките на две години **АКАДЕМИЯТА НА ТЕЛЕРИК** за софтуерни инженери успя да се наложи като безспорен лидер у нас в предлагането на допълнително обучение за софтуерни специалисти, спомагайки за успешния старт в кариерното развитие на стотици ентусиазирани младежи.

academy.telerik.com

academy@telerik.com *facebook.com/TelerikAcademy*

telerik
deliver more than expected

Глава 12. Обработка на изключения

В тази тема...

В настоящата тема ще се запознаем с изключенията в обектно-ориентираното програмиране и в частност в езика C#. Ще се научим как да ги прихващаме чрез конструкцията **`try-catch`**, как да ги предаваме на извикващите методи и как да хвърляме собствени или прихванати изключения чрез конструкцията **`throw`**. Ще дадем редица примери за използването на изключения. Ще разгледаме типовете изключения и йерархията, която образуват в .NET Framework. Накрая ще се запознаем с предимствата при използването на изключения и с това как най-правилно да ги прилагаме в конкретни ситуации.

Какво е изключение?

Докато програмираме ние описваме постъпково какво трябва да направи компютъра (поне в императивното програмиране, за което става дума в тази книга, е така) и в повечето случаи разчитаме на нормалното изпълнение на програмата. В по-голямата част от времето програмите следват този нормален ход на изпълнение, но съществуват и изключения от това правило. Да приемем, че искаме да прочетем файл и да покажем съдържанието му на екрана. Нека файлът се намира на отдалечен сървър и нека по време на отварянето се случи така, че връзката до този сървър пропадне и файлът се зареди само отчасти. Програмата няма да може да се изпълни нормално и да покаже съдържанието на целия файл на екрана. В този случай имаме изключение от правилното (нормалното) изпълнение на програмата и за него трябва да се сигнализира на потребителя и/или администратора.

Изключения

Изключение (exception) в програмирането в общия случай представлява уведомление за дадено събитие, нарушаващо нормалната работа на една програма. Изключенията дават възможност необичайните събития да бъдат обработвани и програмата да реагира на тях по някакъв начин. Когато възникне изключение, конкретното състояние на програмата се запазва и се търси **обработчик на изключението (exception handler)**.

Изключенията се предизвикват или **"хвърлят" (throw an exception)** от програмен код, който трябва да сигнализира на изпълняващата се програма за грешка или необичайна ситуация. Например ако се опитваме да отворим файл, който не съществува, кодът, който отваря файла, ще установи това и ще хвърли изключение с подходящо съобщение за грешка.

Изключенията са една от основните парадигми на обектно-ориентираното програмиране, което е описано подробно в темата "Принципи на обектно-ориентираното програмиране".

Прихващане и обработка на изключения

Exception handling (**инфраструктура за обработка на изключенията**) е механизъм, който позволява хвърлянето и прихващането на изключения. Този механизъм се предоставя от средата за контролирано изпълнение на .NET код, наречена CLR. Част от тази инфраструктура са дефинираните езикови конструкции в C# за хвърляне и прихващане на изключения. CLR се грижи и затова след като веднъж е възникнало всяко изключение да стигне до кода, който може да го обработи.

Изключенията в ООП

В обектно-ориентираното програмиране (ООП) изключенията представляват мощно средство за **централизирана обработка на грешки** и изключителни (необичайни) ситуации. Те заместват в голяма степен процедурно-ориентирания подход за обработка на грешки, при който всяка функция връща като резултат от изпълнението си код на грешка (или неутрална стойност, ако не е настъпила грешка).

В ООП кодът, който извършва дадена операция, обикновено предизвиква изключение, когато в него възникне проблем и операцията не може да бъде изпълнена успешно. Методът, който извиква операцията може да прихване изключението и да обработи грешката или да пропусне изключението и да остави то да бъде прихванато от извикващият го метод. Така грешките не е задължително да бъдат обработвани непосредствено от извикващия код, а могат да се оставят за тези, които са го извикали. Това дава възможност управлението на грешките и необичайните ситуации да се извършва на много нива.

Друга основна концепция при изключенията е тяхната **йерархична същност**. Изключенията в ООП са класове и като такива могат да образуват йерархии посредством наследяване. При прихващането на изключения може да се обработват наведнъж цял клас от грешки, а не само дадена определена грешка (както е в процедурното програмиране).

В ООП се препоръчва чрез изключения да се управлява всяко състояние на грешка или неочаквано поведение, възникнало по време на изпълнението на една програма. Механизмът на изключенията в ООП замества процедурния подход за обработка на грешки и дава много важни предимства като централизирана обработка на грешките, обработка на много грешки на веднъж, възможност за прехвърляне на грешки от даден метод, към извикващия го метод, възможност грешките да се самоописват и да образуват йерархии и обработка на грешките на много нива.

Понякога изключенията се използват за очаквани събития, а не само в случай на проблем, което не е много правилно. Кое е очаквано и кое неочаквано събитие е описано към края на тази глава.

Изключенията в .NET

Изключение (exception) в .NET представлява събитие, което уведомява програмиста, че е възникнало обстоятелство (грешка), непредвидено в нормалния ход на програмата. Това става като методът, в който е възникнала грешката изхвърля специален обект съдържащ информация за вида на грешката, мястото в програмата, където е възникнала, и състоянието на програмата в момента на възникване на грешката.

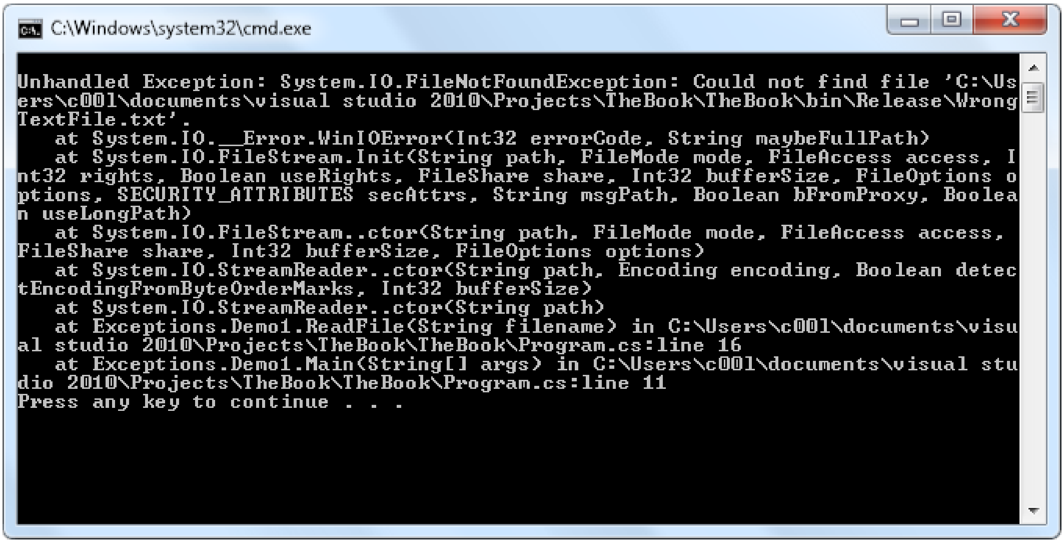
Всяко изключение в .NET носи т.нар. **stack trace** (няма да се мъчим да го превеждаме), който информация за това къде точно в кода е възникнала грешката. Ще го дискутираме подробно малко по-късно.

Пример за код, който хвърля изключения

Типичен пример за код, който хвърля изключения е следният код:

```
class Demo1
{
   static void Main()
   {
      string filename = "WrongTextFile.txt";
      ReadFile(filename);
   }

   static void ReadFile(string filename)
   {
      TextReader reader = new StreamReader(filename);
      string line = reader.ReadLine();
      Console.WriteLine(line);
      reader.Close();
   }
}
```

В примера е даден код, който се опитва да отвори текстов файл и да прочете първия ред от него. Повече за работата с файлове ще научите в главата "Текстови файлове". За момента, нека се съсредоточим не в класовете и методите за работа с файлове, а в конструкциите за работа с изключения. Резултатът от изпълнението на програмата е следният:



```
Unhandled Exception: System.IO.FileNotFoundException: Could not find file 'C:\Us
ers\c001\documents\visual studio 2010\Projects\TheBook\TheBook\bin\Release\Wrong
TextFile.txt'.
   at System.IO.__Error.WinIOError(Int32 errorCode, String maybeFullPath)
   at System.IO.FileStream.Init(String path, FileMode mode, FileAccess access, I
nt32 rights, Boolean useRights, FileShare share, Int32 bufferSize, FileOptions o
ptions, SECURITY_ATTRIBUTES secAttrs, String msgPath, Boolean bFromProxy, Boolea
n useLongPath)
   at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access,
FileShare share, Int32 bufferSize, FileOptions options)
   at System.IO.StreamReader..ctor(String path, Encoding encoding, Boolean detec
tEncodingFromByteOrderMarks, Int32 bufferSize)
   at System.IO.StreamReader..ctor(String path)
   at Exceptions.Demo1.ReadFile(String filename) in C:\Users\c001\documents\visu
al studio 2010\Projects\TheBook\TheBook\Program.cs:line 16
   at Exceptions.Demo1.Main(String[] args) in C:\Users\c001\documents\visual stu
dio 2010\Projects\TheBook\TheBook\Program.cs:line 11
Press any key to continue . . .
```

Първите два реда на метода **`ReadFile()`** съдържат код, в които се хвърлят изключения. В примера конструкторът **`StreamReader(string fileName)`** хвърля **`FileNotFoundException`**, ако не съществува файл с име, каквото му се подава. Методите на потоците, като например **`ReadLine()`**, хвърлят

IOException ако възникне неочакван проблем при входно-изходните операции.

Кодът от примера ще се компилира, но при изпълнение (at run-time) ще хвърли изключение, защото файлът `WrongTextFile.txt` не съществува. Крайният резултат от грешката в този случай е съобщение за грешка, изписано на конзолата, заедно с обяснения къде и как е възникнала тази грешка.

Как работят изключенията?

Ако по време на нормалния ход на програмата някой от извикваните методи неочаквано хвърли изключение, то нормалният ход на програмата се преустановява. Това ще се случи, ако например възникне изключение от типа **FileNotFoundException** при инициализиране на файловия поток от горния пример. Нека разгледаме следния програмен ред:

```
TextReader reader = new StreamReader("WrongTextFile.txt");
```

Ако се случи изключение в този ред, променливата **reader** няма да бъде инициализирана и ще остане със стойност **null** и нито един от следващите редове след този ред от метода няма да бъде изпълнен. Програмата ще преустанови своя ход докато средата за изпълнение CLR не намери обработчик на възникналото изключение **FileNotFoundException**.

Прихващане на изключения в C#

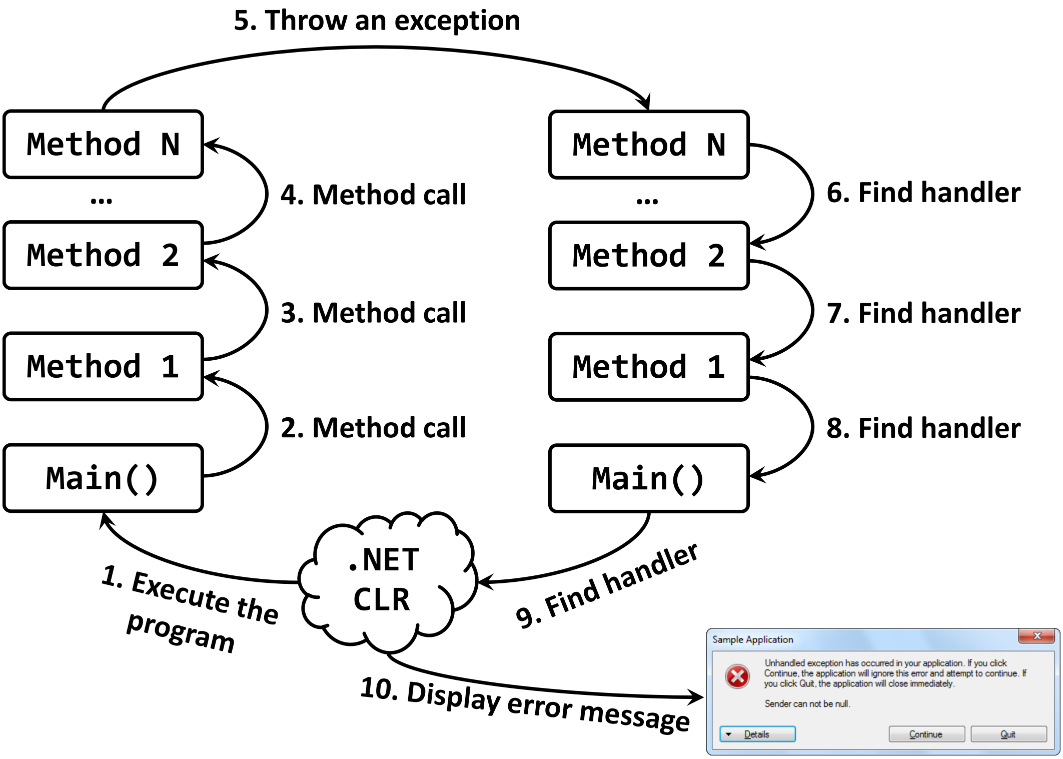
След като един метод хвърли изключение, средата за изпълнение търси код, който евентуално да го прихване и обработи. За да разберем как действа този механизъм ще разгледаме понятието **стек на извикване на методите**. Това е същият този стек, в който се записват всички променливи в програмата, параметрите на методите и стойностните типове.

Всяка програма на .NET започва с **Main(…)** метод. В него може да се извика друг метод – да го наречем "Метод 1", който от своя страна извиква "Метод 2" и т.н., докато се извика "Метод N".

Когато "Метод N" свърши работата си, управлението на програмата се връща към предходния метод и т. н., докато се стигне до **Main(…)** метода. След като се излезе от него, завършва и цялата програма.

Общият принцип е, че когато се извиква нов метод той се добавя най-отгоре в стека, а като завърши изпълнението му, той се изважда от стека. Така в стека за изпълнение на програмата във всеки един момент стоят всички методи, извикани един от друг – от началния метод **Main()** до най-последния извикан метод, който в този момент се изпълнява.

Можем да визуализираме този процес на извикване на методите един от друг по следния начин (стъпки от 1 до 5):


5. Throw an exception
Method N
...
Method 2
Method 1
Main()
4. Method call
3. Method call
2. Method call
Method N
...
Method 2
Method 1
Main()
6. Find handler
7. Find handler
8. Find handler
.NET CLR
1. Execute the program
9. Find handler
10. Display error message
Sample Application
Details
Continue
Quit

Unhandled exception has occurred in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will close immediately.

Sender can not be null.

Процесът на търсене и прихващане на изключение е обратният на този за извикване на методи. Започва се от метода, в който е възникнало изключението и се върви в обратна посока докато се намери метод, където изключението е прихванато (стъпки от 6 до 10). Ако не бъде намерен такъв метод, изключението се прихваща от CLR, който показва съобщение за грешка (изписва я в конзолата или я показва в специален прозорец).

Програмна конструкция try-catch

За да прихванем изключение, обгръщаме парчето код, където може да възникне изключение, с програмната конструкция **`try-catch`**:

```
try
{
   // Some code that may throw an exception
}
catch (ExceptionType objectName)
{
   // Code handling an Exception
}
catch (ExceptionType objectName)
{
   // Code handling an Exception
}
```

Конструкцията се състои от един **`try`** блок, обгръщащ валидни конструкции на C#, които могат да хвърлят изключения, следван от един или няколко **`catch`** блока, които обработват съответно различни по тип изключения. В **`catch`** блока **`ExceptionType`** трябва да е тип на клас, който е наследник на класа **`System.Exception`**. В противен случай ще получим проблем при компилация. Изразът в скобите след **`catch`** играе роля на декларация на променлива и затова вътре в блока **`catch`** можем да използваме обекта **`objectName`**, за да извикваме методите или да използваме свойствата на изключението.

Прихващане на изключения – пример

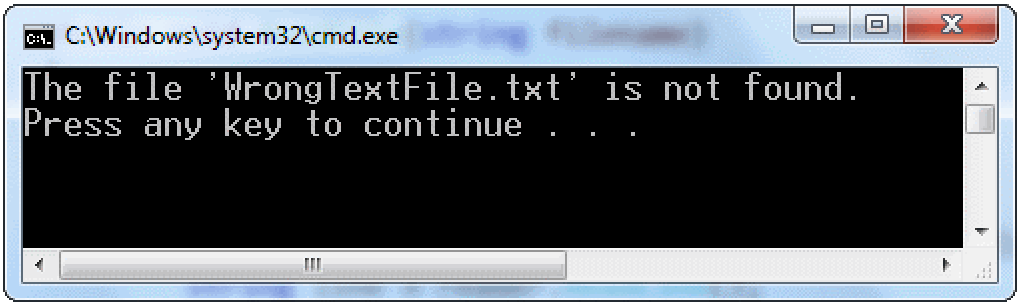
Нека сега направим така, че методът в горния пример сам да обработва изключенията си. За целта заграждаме целия проблемен код, където могат да се хвърлят изключения с **`try-catch`** блок и добавяме прихващане на двата вида изключения:

```
static void ReadFile(string filename)
{
  // Exceptions could be thrown in the code below
  try
  {
    TextReader reader = new StreamReader(filename);
    string line = reader.ReadLine();
    Console.WriteLine(line);
    reader.Close();
  }
  catch (FileNotFoundException fnfe)
  {
    // Exception handler for FileNotFoundException
    // We just inform the user that there is no such file
    Console.WriteLine("The file '{0}' is not found.", filename);
  }
  catch (IOException ioe)
  {
    // Exception handler for other input/output exceptions
    // We just print the stack trace on the console
    Console.WriteLine(ioe.StackTrace);
  }
}
```

Добре, сега методът работи по малко по-различен начин. При възникване на **`FileNotFoundException`** по време на изпълнението на конструкцията **`new StreamReader(string fileName)`** **средата за изпълнение (Common Language Runtime - CLR)** няма да изпълни следващите редове, а ще прескочи чак на реда, където изключението е прихванато с конструкцията **`catch (FileNotFoundException fnfe)`**:

```
catch (FileNotFoundException fnfe)
{
  // Exception handler for FileNotFoundException
  // We just inform the user that there is no such file
  Console.WriteLine("The file '{0}' is not found.", filename);
}
```

Като обработка на изключението потребителите просто ще бъдат информирани, че такъв файл не съществува. Това се извършва чрез съобщение, изведено на стандартния изход:



```
The file 'WrongTextFile.txt' is not found.
Press any key to continue . . .
```

Аналогично, ако възникне изключение от тип **`IOException`** по време на изпълнението на метода **`reader.ReadLine()`**, то се обработва от блока:

```
catch (IOException ioe)
{
  // Exception handler for FileNotFoundException
  // We just print the stack trace on the screen
  Console.WriteLine(ioe.StackTrace);
}
```

Понеже не знаем естеството на грешката, породила грешно четене, отпечатваме цялата информация за изключението на стандартния изход.

Редовете код между мястото на възникване на изключението и мястото на прихващане и обработка не се изпълняват.

Отпечатването на цялата информация от изключението (stack trace) на потребителя не винаги е добра практика! Как най-правилно се обработват изключения е описано в частта за добри практики.

Stack Trace

Информацията, която носи т. нар. **stack trace**, съдържа подробно описание на естеството на изключението и на мястото в програмата, където то е възникнало. Stack trace се използва от програмистите, за да се намерят причините за възникването на изключението. Stack trace съдържа голямо количество информация и е предназначен за анализиране само от програмистите и администраторите, но не и от крайните потребители на програмата, които не са длъжни да са технически лица. Stack trace е стандартно средство за търсене и отстраняване (дебъгване) на проблеми.

Stack Trace – пример

Ето как изглежда stack trace на изключение за липсващ файл от първия пример (без try-catch клаузите):

```
Unhandled Exception: System.IO.FileNotFoundException: Could not
find file '…\WrongTextFile.txt'.
   at System.IO.__Error.WinIOError(Int32 errorCode, String
maybeFullPath)
   at System.IO.FileStream.Init(String path, FileMode mode,
FileAccess access, Int32 rights, Boolean useRights, FileShare
share, Int32 bufferSize, FileOptions options,
SECURITY_ATTRIBUTES secAttrs, String msgPath, Boolean
bFromProxy, Boolean useLongPath)
   at System.IO.FileStream..ctor(String path, FileMode mode,
FileAccess access, FileShare share, Int32 bufferSize,
FileOptions options)
   at System.IO.StreamReader..ctor(String path, Encoding
encoding, Boolean detectEncodingFromByteOrderMarks, Int32
bufferSize)
   at System.IO.StreamReader..ctor(String path)
   at Exceptions.Demo1.ReadFile(String filename) in
Program.cs:line 17
   at Exceptions.Demo1.Main() in Program.cs:line 11

Press any key to continue . . .
```

Системата не може да намери този файл и затова, за да съобщи за възникващ проблем хвърля изключението **`FileNotFoundException`**.

Как да разчетем "Stack Trace"?

За да се ориентираме в един stack trace трябва да можем да го разчетем правилно и да знаем неговата структура.

Stack trace съдържа следната информация в себе си:

- Пълното име на класа на изключението;
- Съобщение – информация за естеството на грешката;
- Информация за стека на извикване на методите.

От примера по-горе пълното име на изключението е **`System.IO.FileNotFoundException`**. Следва съобщението за грешка. То донякъде повтаря името на самото изключение: "**`Could not find file '…\WrongTextFile.txt'.`**". Следва целият стек на извикване на методите, който по традиция е най-дългата част от всеки stack trace. Един ред от стека съдържа нещо такова:

```
at <namespace>.<class>.<method> in <source file>.cs:line <line>
```

Всички методи от стека на извикванията са показани на отделен ред. Най-отгоре (на върха на стека) е методът, който първоначално е хвърлил изключение, а най-отдолу е **`Main()`** методът (на дъното на стека). Всеки метод се дава заедно с класа, който го съдържа и в скоби реда от файла (ако сорс кодът е наличен), където е хвърлено изключението, примерно:

```
   at Exceptions.Demo1.ReadFile(String filename) in
…\Program.cs:line 17
```

Редовете са налични само ако класът е компилиран с опция да включва дебъг информация (тя включва номера на редове, имена на променливи и друга информация, спомагаща дебъгването на програмата). Дебъг информацията се намира извън .NET асемблитата, в т.нар. debug symbols file (**.pdb**). Както се вижда от примерния stack trace, за някои асемблита е налична дебъг информация и се извеждат номерата на редовете от стека, а за други (например системните асемблита от .NET Framework) такава информация липсва и не е ясно на кой ред и в кой файл със сорс код е възникнала проблемната ситуация.

Ако методът е конструктор, то вместо името му се изписва служебното наименование **`.ctor`**, например: **`System.IO.StreamReader..ctor(String path)`**. Ако липсва информация за сорс файла и номера на реда, където е възникнало изключението, не се изписва име на файл и номер на ред.

Това позволява бързо и лесно да се намери класът, методът и дори редът, където е възникнала грешката, да се анализира нейното естество и да се поправи.

Хвърляне на изключения (конструкцията throw)

Изключения в C# се хвърлят с ключовата дума **`throw`**, като първо се създава инстанция на изключението и се попълва нужната информация за него. Изключенията са обикновени класове, като единственото изискване за тях е да наследяват **`System.Exception`**.

Ето един пример:

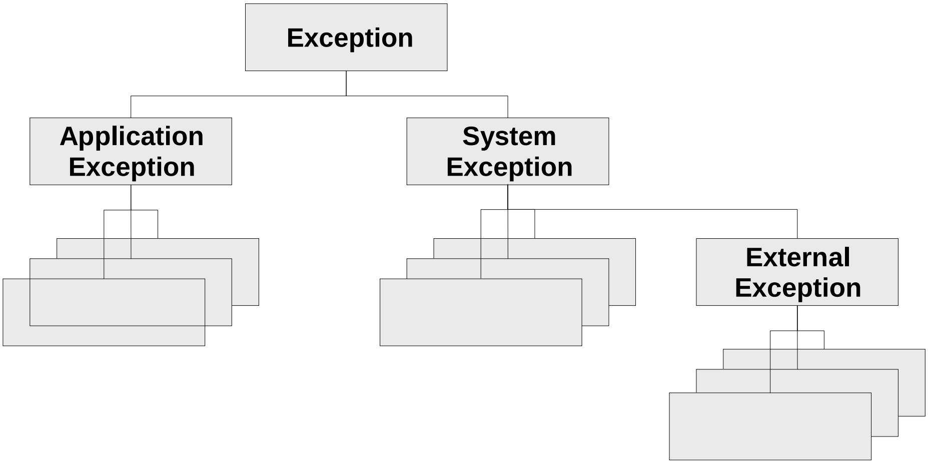
```
static void Main()
{
   Exception e = new Exception("There was a problem");
   throw e;
}
```

Резултатът от изпълнението на програмата е следният:

```
Unhandled Exception: System.Exception: There was a problem
   at Exceptions.Demo1.Main() in Program.cs:line 11
Press any key to continue . . .
```

Йерархия на изключенията

В .NET Framework има два типа от изключения: изключения генерирани от дадена програма (`ApplicationException`) и изключения генерирани от средата за изпълнение (`SystemException`). Всяко едно от тези изключения включва собствена йерархия от изключения-наследници.



Тъй като наследниците на всеки от тези класове имат различни характеристики, ще разгледаме всеки от тях поотделно.

Класът Exception

В .NET Framework `Exception` е базовият клас на всички изключения. Няколко класа на изключения го наследяват директно, включително `ApplicationException` и `SystemException`. Тези два класа са базови за почти всички изключения, възникващи по време на изпълнение на програмата.

Класът `Exception` съдържа копие на стека по време на създаването на изключението. Съдържа още кратко текстово съобщение описващо грешката (попълва се от метода, който хвърля изключението). Всяко изключение може да съдържа още **причина (cause)** за възникването му, която представлява друго изключение – оригиналната причина за появата на проблема. Можем да го наричаме **вътрешно (обвито) изключение** (**inner / wrapped exception**) или **вложено изключение**.

Външното изключение се нарича **обгръщащо (обвиващо) изключение**. Така може да се навържат много изключения. В този случай говорим за **верига от изключения (exception chain)**.

Exception – конструктори, методи, свойства

Ето как изглежда класът **`System.Exception`**:

```
[SerializableAttribute]
[ComVisibleAttribute(true)]
[ClassInterfaceAttribute(ClassInterfaceType.None)]
public class Exception : ISerializable, _Exception
{
  public Exception();
  public Exception(string message);
  public Exception(string message, Exception innerException);
  public virtual IDictionary Data { get; }
  public virtual string HelpLink { get; set; }
  protected int HResult { get; set; }
  public Exception InnerException { get; }
  public virtual string Message { get; }
  public virtual string Source { get; set; }
  public virtual string StackTrace { get; }
  public MethodBase TargetSite { get; }
  public virtual Exception GetBaseException();
}
```

Нека обясним накратко по-важните от тези методи, тъй като те се наследяват от всички изключения в .NET Framework:

- Имаме три конструктора с различните комбинации за съобщение и обвито изключение.
- Свойството **`Message`** връща текстово описание на изключението. Например, ако изключението е **`FileNotFoundException`**, то описанието може да обяснява кой точно файл не е намерен. Всяко изключение само решава какво съобщение за грешка да върне. Най-често се позволява на хвърлящият изключението код да подаде това описание на конструктора на хвърляното изключение. След като е веднъж зададено, свойството **`Message`** не може повече да се променя.
- Свойството **`InnerException`** връща вътрешното (обвитото) изключение или **`null`**, ако няма такова.
- Методът **`GetBaseException()`** връща най-вътрешното изключение. Извикването на този метод за всяко изключение от една верига изключения трябва да върне един и същ резултат – изключението, което е възникнало първо.
- Свойството **`StackTrace`** връща информация за целия стек, който се пази в изключението (вече видяхме как изглежда тази информация).

Application vs. System Exceptions

Изключенията в .NET Framework са два вида – системни и потребителски. Системните изключения са дефинирани в библиотеките от .NET Framework и се ползват вътрешно от него, а потребителските изключения се дефинират от програмиста и се използват от софтуера, по който той работи. При разработката на приложение, което хвърля собствени изключения, е добра практика тези изключения да наследяват **`Exception`**. Наследяването на класа **`SystemException`** би трябвало да става само вътрешно от .NET Framework.

Най-тежките изключения – тези хвърляни от средата за изпълнение – включват **`ExecutionEngineException`** (вътрешна грешка при работата на CLR), **`StackOverflowException`** (препълване на стека, най-вероятно заради бездънна рекурсия) и **`OutOfMemoryException`** (препълване на паметта). И при трите изключения възможностите за адекватна реакция от страна на вашата програма са минимални. На практика тези изключения означават фатално счупване (crash) на приложението.

Изключенията при взаимодействие с външни за средата за изпълнение компоненти наследяват **`ExternalException`**. Такива са **`COMException`**, **`Win32Exception`** и **`SEHException`**.

Хвърляне и прихващане на изключения

Нека разгледаме в детайли някои особености при хвърлянето и прихващането на изключения.

Вложени (nested) изключения

Вече споменахме, че в едно изключение може да съдържа в себе си вложено (опаковано) друго изключение. Защо се налага едно изключение да бъде опаковано в друго? Нека обясним тази често използвана практика при обработката на изключения в ООП.

Добра практика в софтуерното инженерство е всеки модул / компонент / програма да дефинира малък брой **application exceptions** (изключения написани от автора на модула / програмата) и този компонент да се ограничава само до тях, а не да хвърля стандартни .NET изключения, наричани още **системни изключения (system exceptions)**. Така ползвателят на този модул / компонент знае какви изключения могат да възникнат в него и няма нужда да се занимава с технически подробности.

Например един модул, който се занимава с олихвяването в една банка би трябвало да хвърля изключения само от неговата бизнес област, примерно **`InterestCalculationException`** и **`InvalidPeriodException`**, но не и изключения като **`FileNotFoundException`**, **`DivideByZeroException`** и **`NullReferenceException`**. При възникване на някое изключение, което не е свързано директно с проблемите на олихвяването, то се обвива в друго изключение от тип **`InterestCalculationException`** и така извикващия метод получава информация, че олихвяването не е успешно, а като детайли за неуспеха може да разгледа оригиналното изключение, причинител на проблема, от което примерно може да стане ясно, че няма връзка със сървъра за бази данни.

Тези application exceptions от бизнес областта на решавания проблем, за които дадохме пример, обаче не съдържат достатъчно информация за възникналата грешка, за да бъде поправена тя. Затова е добра практика в тях да има и техническа информация за оригиналния причинител на проблема, която е много полезна, например при дебъгване.

Същото обяснение от друга гледна точка: един компонент А има дефинирани малък брой изключения (А-изключения). Този компонент използва друг компонент Б. Ако Б хвърли Б-изключение, то А не може да си свърши работата и също трябва да хвърли изключение, но не може да хвърли Б-изключение, затова хвърля А-изключение, съдържащо изключението Б като вложено изключение.

Защо А не може да хвърли Б-изключение? Има много причини:

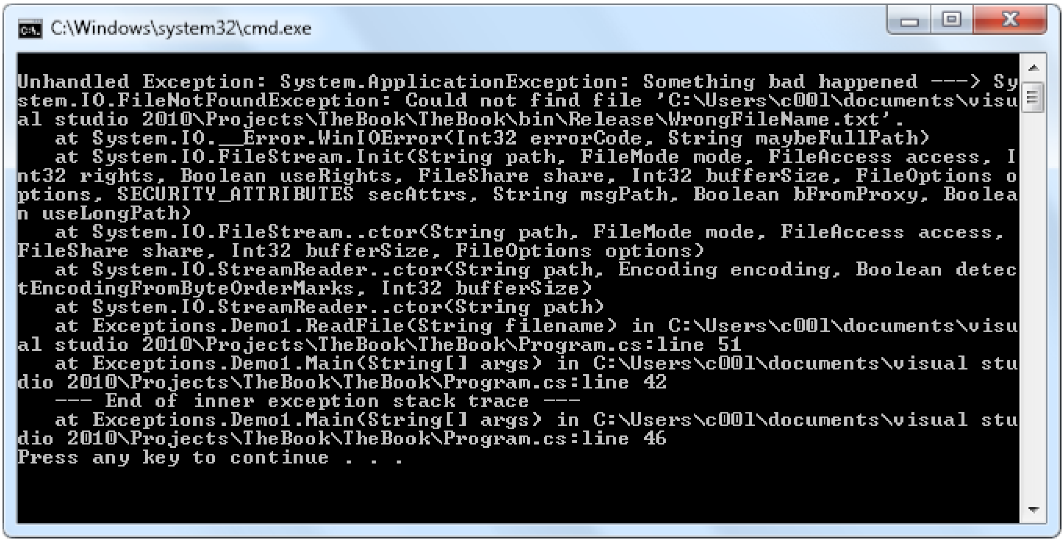
- Ползвателите на А не трябва да знаят за съществуването на Б (за повече информация разгледайте точката за абстракция от главата за принципите на ООП).
- Компонентът А не е дефинирал, че ще хвърля Б-изключения.
- Ползвателите на А не са подготвени за Б-изключения. Те очакват само А-изключения.

Как да разчетем "Stack Trace" на вериги изключения?

Сега ще дадем пример как можем да създадем верига от изключения и ще демонстрираме как се изписва на екрана вложено изключение. Нека имаме следния код (забележете, че вляво са дадени редовете от кода):

```
37  static void Main()
38  {
39     try
40     {
41        string fileName = "WrongFileName.txt";
42        ReadFile(fileName);
43     }
44     catch (Exception e)
45     {
46        throw new ApplicationException("Smth bad happened", e);
47     }
48  }
49  static void ReadFile(string fileName)
50  {
51     TextReader reader = new StreamReader(fileName);
52    string line = reader.ReadLine();
53    Console.WriteLine(line);
54    reader.Close();
55  }
```

В този пример извикваме метода **`ReadFile()`**, който хвърля изключение, защото файлът не съществува. В **`Main()`** метода прихващаме всички изключения, опаковаме ги в наше собствено изключение от тип **`ApplicationException`** и ги хвърляме отново. Резултатът от изпълнението на този код е следният:



```
Unhandled Exception: System.ApplicationException: Something bad happened ---> Sy
stem.IO.FileNotFoundException: Could not find file 'C:\Users\c001\documents\visu
al studio 2010\Projects\TheBook\TheBook\bin\Release\WrongFileName.txt'.
   at System.IO.__Error.WinIOError(Int32 errorCode, String maybeFullPath)
   at System.IO.FileStream.Init(String path, FileMode mode, FileAccess access, I
nt32 rights, Boolean useRights, FileShare share, Int32 bufferSize, FileOptions o
ptions, SECURITY_ATTRIBUTES secAttrs, String msgPath, Boolean bFromProxy, Boolea
n useLongPath)
   at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access,
FileShare share, Int32 bufferSize, FileOptions options)
   at System.IO.StreamReader..ctor(String path, Encoding encoding, Boolean detec
tEncodingFromByteOrderMarks, Int32 bufferSize)
   at System.IO.StreamReader..ctor(String path)
   at Exceptions.Demo1.ReadFile(String filename) in C:\Users\c001\documents\visu
al studio 2010\Projects\TheBook\TheBook\Program.cs:line 51
   at Exceptions.Demo1.Main(String[] args) in C:\Users\c001\documents\visual stu
dio 2010\Projects\TheBook\TheBook\Program.cs:line 42
   --- End of inner exception stack trace ---
   at Exceptions.Demo1.Main(String[] args) in C:\Users\c001\documents\visual stu
dio 2010\Projects\TheBook\TheBook\Program.cs:line 46
Press any key to continue . . .
```

Нека се опитаме заедно да проследим редовете от stack trace в сорс кода. Забелязваме, че се появява секция, която описва край на вложеното изключение:

```
--- End of inner exception stack trace ---
```

Това ни дава полезна информация за това как се е стигнало до хвърлянето на изключението, което разглеждаме.

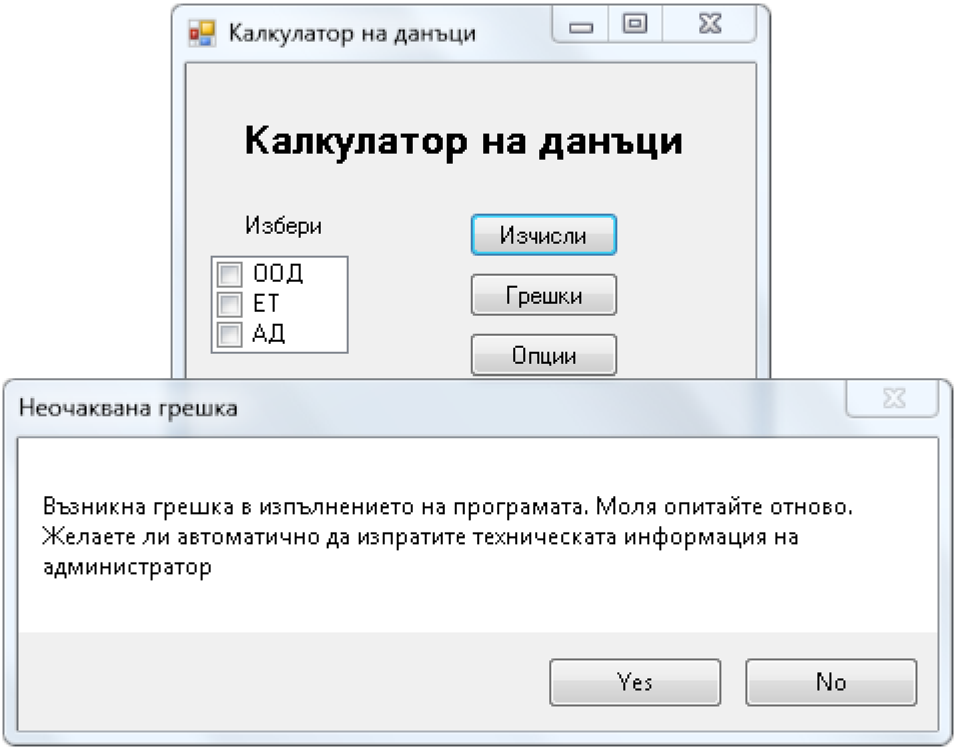
Забележете първия ред. Той има следния вид:

```
Unhandled Exception: Exception1: Msg1 ---> Exception2: Msg2
```

Това показва, че изключение от тип **`Exception1`** е обвило изключение от тип **`Exception2`**. След всяко изключение се изписва и съответното му съобщение за грешка (свойството **`Message`**). Всеки метод от стека съдържа името на файла, в който е възникнало съответното изключение и номера на реда. Може да проследим по номерата на редовете от примера къде и как точно са възникнали изключенията, отпечатани на конзолата.

Визуализация на изключения

Във визуалните (GUI) приложения изключенията, които не могат да бъдат обработени (или най-общо казано грешките), трябва да се показват на потребителя под формата на диалогов прозорец съдържащ описание, съобразено с познанията на потребителите:



Възникна грешка в изпълнението на програмата. Моля опитайте отново. Желаете ли автоматично да изпратите техническата информация на администратор

В конзолните приложения най-често грешката се изписва на конзолата във вид на stack trace, въпреки че това може и да не е най-удобния за крайния потребител начин за уведомление за проблеми.

При уеб приложения грешката се визуализира като червен текст в началото на страницата или около след полето, за което се отнася.

Както можете сами да си направите извода, при различните приложения изключенията и грешките се обработват по различни начини. По тази причина има много препоръки кои изключения да се хванат и кои не и как точно да се визуализират съобщенията за грешки, за да не се стряскат потребителите. Нека обясним някои от тези препоръки.

Кои изключения да обработим и кои не?

Има едно универсално правило за обработката на изключенията:

⚠	**Един метод трябва да обработва само изключенията, за които е компетентен, които очаква и за които има знания**

	как да ги обработи. Останалите трябва да изхвърля към извикващия метод.

Ако изключенията се предават по гореописания начин от метод на метод и не се прихванат никъде, те неминуемо ще достигнат до началния метод от програмата – **`Main()`** метода – и ако и той не ги прихване, средата за изпълнение ще ги отпечата на конзолата (или ще ги визуализира по друг начин, ако няма конзола) и ще преустанови изпълнението на програмата.

Какво означава един метод да е "компетентен, за да обработи да едно изключение"? Това означава, че той очаква това изключение и знае кога точно може да възникне и знае как да реагира в този специален случай. Ето един пример. Имаме метод, който трябва да прочете даден текстов файл, а ако файлът не съществува, трябва да върне празен низ. Този метод би могъл да прихване съобщението **`FileNotFoundException`** и да го обработи. Той знае какво да прави, когато файлът липсва – трябва да върне празен низ. Какво става, обаче, ако при отварянето на файла се получи **`OutOfMemoryException`**? Компетентен ли е методът да обработи тази ситуация? Как може да я обработи? Дали трябва да върне празен низ, дали трябва да хвърли друго изключение или да направи нещо друго? Очевидно методът за четене на файл не е компетентен да се справи със ситуацията "недостиг на памет" и най-доброто, което може да направи е да остави изключението необработено. Така то може да бъде прихванато на друго ниво от някой по-компетентен метод. Това е цялата идея: всеки метод прихваща изключенията, от които разбира, а останалите ги остава на останалите методи. Така методите си поделят по ясен и систематичен начин отговорностите.

Изхвърляне на изключения от Main() метода – пример

Изхвърлянето на изключения от **`Main()`** метода по принцип не е желателно. Вместо това се препоръчва всички изключения да бъдат прихванати и обработени. Изхвърлянето на изключения от **`Main()`** метода все пак е възможно, както от всеки друг метод:

```
static void Main()
{
  throw new Exception("Ooops!");
}
```

Всички изключения изхвърлени от **`Main()`** метода се прихващат от самата среда за изпълнение (.NET CLR) и се обработват по един и същ начин – пълният stack trace на изключението се изписва на конзолата или се визуализира по друг начин. Такова изхвърляне на изключенията, възникващи в **`Main()`** метода е много удобно, когато пишем кратка програмка набързо и не искаме да обработваме евентуално възникващите изключения. Това е бягане от отговорност, което се прави при малки прости програмки, но не трябва да се случва при големи и сериозни приложения.

Прихващане на изключения на нива – пример

Възможността за пропускане на изключения през даден метод ни позволява да разгледаме един по-сложен пример: прихващане на изключения на нива. **Прихващането на нива** е комбинация от прихващането на определени изключения в дадени методи и пропускане на всички останали изключения към предходните методи (нива) в стека. В примера по-долу изключенията възникващи в метода **ReadFile()** се прихващат на две нива (в **try-catch** блока на **ReadFile(…)** метода и в **try-catch** блока на **Main()** метода):

```
static void Main()
{
  try
  {
    string fileName = "WrongFileName.txt";
    ReadFile(fileName);
  }
  catch (Exception e)
  {
    throw new ApplicationException("Bad thing happened", e);
  }
}
static void ReadFile(string fileName)
{
  try
  {
    TextReader reader = new StreamReader(fileName);
    string line = reader.ReadLine();
    Console.WriteLine(line);
    reader.Close();
  }
  catch (FileNotFoundException fnfe)
  {
    Console.WriteLine("The file {0} does not exist!",
      filename);
  }
}
```

Първото ниво на прихващане на изключенията в примера е в метода **ReadFile()**, а второто ниво е в **Main()** метода. Методът **ReadFile()** прихваща само изключенията от тип **FileNotFoundException**, а пропуска всички останали **IOException** изключения към **Main()** метода, където те биват прихванати и обработени. Всички останали изключения, които не са от групата **IOException** (например **OutOfMemoryException**) не се прихващат на никое от двете нива и се оставят на CLR да се погрижи за тях.

Ако **`Main()`** методът подаде име на несъществуващ файл то ще възникне **`FileNotFoundException`**, което ще се прихване в **`ReadFile()`**. Ако обаче се подаде име на съществуващ файл и възникне грешка при самото четене на файла (например няма права за достъп до файла), то изключението ще се прихване в **`Main()`** метода.

Прихващането на изключения на нива позволява отделните изключения да се обработват на най-подходящото място. Така се постига огромна гъвкавост, удобство и чистота на кода, отговорен за обработка на проблемните ситуации в програмата.

Конструкцията try-finally

Всеки блок **`try`** може да съдържа блок **`finally`**. Блокът **`finally`** се изпълнява **винаги** при излизане от **`try`** блока, независимо как се излиза от **`try`** блока. Това гарантира изпълнението на **`finally`** блока, дори ако възникне неочаквано изключение или методът завърши с израз **`return`**.

Блокът `finally` няма да се изпълни, ако по време на изпълнението на блока `try` средата за изпълнение CLR прекрати изпълнението си!

Блокът **`finally`** има следната основна форма:

```
try {
   Some code that could or could not cause an exception
} finally {
   // Code here will allways execute
}
```

Всеки **`try`** блок може да има нула или повече **`catch`** блокове и максимум един блок **`finally`**. Възможна е и комбинация с множество **`catch`** блокове и един **`finally`** блок:

```
try {
   some code
} catch (…) {
   // Code handling an exception
} catch (…) {
   // Code handling another exception
} finally {
   // This code will allways execute
}
```

Кога да използваме try-finally?

В много приложения се налага да се работи с външни за програмата ресурси: файлове, мрежови връзки, графични елементи от операционната система, комуникационни канали (pipes), потоци от и към различни периферни устройства (принтер, звукова карта, карточетец и други). При работата с външни ресурси е важно след като веднъж е заделен даден ресурс, той да бъде освободен възможно най-скоро след като вече не е нужен на програмата. Например, ако отворим някакъв файл, за да прочетем съдържанието му (примерно за да заредим JPEG картинка), е важно да го затворим веднага след като го прочетем. Ако оставим файла отворен, това ограничава достъпа на останалите потребители като забранява някои операции, например промяна на файла и изтриване. Може би ви се е случвало да не можете да изтриете дадена директория с файлове, нали? Най-вероятната причина за това е, че някой от файловете в директорията е отворен в момента от друго приложение и така изтриването му е блокирано от операционната система.

Блокът `finally` е незаменим при нужда от освобождаване на вече заети ресурси. Ако го нямаше, никога не бихме били сигурни дали разчистването на заделените ресурси няма случайно да бъде прескочено при неочаквано изключение или заради използването на някой от изразите `return`, `continue` или `break`.

Тъй като концепцията за правилно заделяне и освобождаване на ресурси е важна за програмирането (независимо от езика и платформата), ще обясним подробно и ще илюстрираме с примери как се прави това.

Освобождаване на ресурси – дефиниране на проблема

В примера, който разглеждаме, искаме да прочетен даден файл. Имаме четец (reader), който задължително трябва да се затвори след като файлът е прочетен. Най-правилният начин това да се направи е с `try-finally` блок ограждащ редовете, където се използват съответните потоци. Да си припомним примера:

```
static void ReadFile(string fileName)
{
  TextReader reader = new StreamReader(fileName);
  string line = reader.ReadLine();
  Console.WriteLine(line);
  reader.Close();
}
```

Какъв е проблемът с този код? Той би трябвало да отваря файлов четец, да чете данни от него и накрая следва задължително да затвори файла преди да завърши изпълнението на метода. Задължителното затваряне на файловете е проблемна ситуация, защото от метода може да се излезе по няколко начина:

- По време на инициализиране на четеца може да възникне непредвидено изключение (например ако липсва файлът).

- По време на четенето на данните може възникне непредвидено изключение (например ако файлът се намира на отдалечено мрежово устройство, до което бъде изгубена връзката).
- Между инициализирането и затварянето на потоците се изпълни операторът **`return`**.
- Всичко е нормално и не възникват никакви изключения.

Така написан примерният код за четене на файл е логически грешен, защото четецът ще се затвори правилно само в последния случай (ако не възникнат никакви изключения). Във всички останали случаи четецът няма да се затвори, защото ще възникне изключение и кодът за затваряне на файла няма да се извика. Имаме проблем, макар и да не взимаме под внимание възможността отварянето, използването и затварянето на потока да е част от тяло на цикъл, където може да се използват изразите **`continue`** и **`break`**, което също ще доведе до незатваряне на потоците.

Освобождаване на ресурси – решение на проблема

Демонстрирахме, че схемата "отваряме файл, четем го, затваряме го" концептуално е грешна, защото ако при четенето възникне изключение, файлът ще си остане отворен. Как тогава трябва да напишем кода, така че файлът да се затваря правилно във всички ситуации.

Всички тези главоболия можем да си спестим като използваме конструкцията **`try-finally`**. Ще разгледаме първо пример с един ресурс (в случая файл), а след това и с два и повече ресурса.

Сигурното затваряне на файл (поток) може да се извърши по следния начин:

```
static void ReadFile(string fileName)
{
   TextReader reader = null;
   try
   {
      reader = new StreamReader(fileName);
      string line = reader.ReadLine();
      Console.WriteLine(line);
   }
   finally
   {
      // Always close "reader" (first check if properly opened)
      if (reader != null)
      {
         reader.Close();
      }
   }
}
```

Да анализираме примера. Първоначално декларираме променлива **`reader`** от тип **`TextReader`**, след това отваряме **`try`** блок, в който инициализираме нов четец, използваме го и накрая го затваряме във **`finally`** блок. Каквото и да стане при използването и инициализацията, сме сигурни, че четецът и свързания с него файл ще бъдат затворени. Ако има проблем при инициализацията, например липсващ файл, то ще се хвърли **`FileNotFoundException`** и променливата **`reader`** ще остане със стойност **`null`**. За този случай и за да се избегне **`NullReferenceException`** е необходимо да се прибави проверка дали **`reader`** не е **`null`** преди да се извика методът **`Close()`** за затваряне на четеца. Ако имаме **`null`**, то четецът изобщо не е бил инициализиран и няма нужда да бъде затварян. При всички сценарии на изпълнение (при нормално четене, при грешка или при някакъв друг проблем) се гарантира, че ако файлът е бил отворен, той ще бъде съответно затворен преди излизане от метода.

Горният пример трябва подходящо да обработи всички изключения, които възникват при инициализиране (**`FileNotFoundException`**) и използване на четеца. В примера възможните изключения просто се изхвърлят от метода, тъй като той не е компетентен да ги обработи.

Даденият пример е за файлове (потоци), но може да се използва за произволни ресурси, които изискват задължително освобождаване след приключване на работата с тях. Такива ресурси могат да бъдат връзки към отдалечени компютри, връзки с бази данни, обекти от операционната система и други.

Освобождаване на ресурси – алтернативно решение

Горната конструкция е вярна, но е излишно сложна. Нека разгледаме един неин опростен вариант:

```
static void ReadFile(string fileName)
{
   TextReader reader = new StreamReader(fileName);
   try
   {
      string line = reader.ReadLine();
      Console.WriteLine(line);
   }
   finally
   {
      reader.Close();
   }
}
```

Предимството на този вариант е по-краткия запис – спестяваме една излишна декларация на променливата **`reader`** и избягваме проверката за **`null`**. Проверката за **`null`** е излишна, защото инициализацията на потока е извън **`try`** блока и ако е възникнало изключение докато тя се изпълнява изобщо няма да се стигне до изпълнение на **`finally`** блока и затварянето на потока.

Този вариант е по-чист, по-кратък и по-ясен и е известен като **шаблон за освобождаване на ресурси (dispose pattern)**.

Освобождаване на множество ресурси

Досега разгледахме използването на **`try-finally`** за освобождаване на само един ресурс, но понякога може да има нужда да се освободят повече от един ресурс. Добра практика е ресурсите да се освобождават в ред обратен на този на заделянето им.

За освобождаването на множество ресурси могат да се използват горните два подхода като **`try-finally`** блоковете се влагат един в друг:

```
static void ReadFile(string filename)
{
   Resource r1 = new Resource1();
   try
   {
      Resource r2 = new Resource2();
      try
      {
         // Use r1 and r2
      }
      finally
      {
         r2.Release();
      }
   }
   finally
   {
      r1.Release();
   }
}
```

Другият вариант е всички ресурси да се декларират предварително и накрая да се освободят в един единствен **`finally`** блок с проверка за **`null`**:

```
static void ReadFile(string filename)
{
   Resource r1 = null;
   Resource r2 = null;
   try
   {
      Resource r1 = new Resource1();
```

```
    Resource r2 = new Resource2();

    // Use r1 and r2
  }
  finally
  {
    r1.Release();
    r2.Release();
  }
}
```

И двата подхода са правилни със съответните предимства и недостатъци и се прилагат в зависимост от предпочитанията на програмиста съобразно конкретната ситуация. Все пак вторият подход е малко рисков, тъй като ако във **`finally`** блока възникне изключение (което почти никога не се случва) при затварянето на първия ресурс, вторият ресурс няма да бъде затворен. При първия подход няма такъв проблем, но се пише повече код.

IDisposable и конструкцията using

Време е да обясним и един съкратен запис в езика C# за освобождаване на някои видове ресурси. Ще покажем кои точно ресурси могат да се възползват от този запис и как точно изглежда той.

IDisposable

Основната употреба на интерфейса **`IDisposable`** е за освобождаване не ресурси. В .NET такива ресурси са графични елементи (window handles), файлове, потоци и др. За интерфейси ще стане дума в главата "Принципи на обектно-ориентираното програмиране", но за момента можете да считате, че интерфейсът е индикация, че даден тип обекти (например потоците за четене на файлове) поддържат определено множество операции (например затваряне на потока и освобождаване на свързаните с него ресурси).

Няма да навлизаме в подробности как се имплементира **`IDisposable`** (нито ще дадем примери), защото ще трябва да навлезем в доста сложна материя и да обясним как работи системата за почистване на паметта (garbage collector) и как се работи с деструктори, неуправлявани ресурси и т.н.

Важният метод в интерфейса **`IDisposable`** е **`Dispose()`**. Основното, което трябва да се знае за него е, че той освобождава ресурсите на класа, който го имплементира. В случая, когато ресурсите са потоци, четци или файлове, освобождаването им може да се извърши с метода **`Dispose()`** от интерфейса **`IDisposable`**, който извиква метода им **`Close()`**, който ги затваря и освобождава свързаните с тях ресурси от операционната система. Така затварянето на един поток може да стане по следния начин:

```
StreamReader reader = new StreamReader(fileName);
try
{
   // Use the reader here
}
finally
{
   if (reader != null)
   {
      reader.Dispose();
   }
}
```

Ключовата дума using

Последният пример може да се запише съкратено с помощта на ключовата дума **`using`** в езика C# по следния начин:

```
using (StreamReader reader = new StreamReader(fileName) )
{
   // Use the reader here
}
```

Определено този вариант изглежда доста по-кратък и по-ясен, нали? Не е нужно нито да имаме **`try-finally`**, нито да викаме изрично някакви методи за освобождаването на ресурсите. Компилаторът се грижи да сложи автоматично **`try-finally`** блок, с който при излизане от using блока, т.е. достигане на неговата затваряща скоба **`}`**, да извика метода **`Dispose()`** за освобождаване на използвания в блока ресурс.

Вложени using конструкции

Конструкциите **`using`** могат да се влагат една в друга:

```
using (ResourceType r1 = …)
   using (ResourceType r2 = …)
      ...
         using (ResourceType rN = …)
            statements;
```

Горният код може да се запише съкратено и по следния начин:

```
using (ResourceType r1 = …, r2 = …, …, rN = …)
{
   statements;
}
```

Важно е да се отбележи, че конструкцията **`using`** няма никакво отношение към изключенията. Нейната единствена роля е да освободи ресурсите без значение дали са били хвърлени изключения или не и какви изключения евентуални са били хвърлени.

Кога да използване using?

Има много просто правило кога трябва да се използва using при работата с някой .NET клас:

Използвайте `using` при работа с всички класове, които имплементират `IDisposable`. Проверявайте за `IDisposable` в MSDN.

Когато даден клас имплементира **`IDisposable`**, това означава, че авторът на този клас е предвидил той да бъде използван с конструкцията **`using`**. Това означава, че този клас обвива в себе си някакъв ресурс, който е ценен и не може да се оставя неосвободен, дори при екстремни условия. Ако даден клас имплементира **`IDisposable`**, значи трябва да се освобождава задължително веднага след като работата с него приключи и това става най-лесно с конструкцията **`using`** в C#.

Предимства при използване на изключения

След като се запознахме подробно с изключенията, техните свойства и с това как да работим с тях, нека разгледаме причините те да бъдат въведени и да придобият широко разпространение.

Отделяне на кода за обработка на грешките

Използването на изключения позволява да се отдели кодът, описващ нормалното протичане на една програма, от кода необходим в изключителни ситуации и кода необходим при обработване на грешки. Това ще демонстрираме със следния пример, който е приблизителен псевдокод на примера разгледан от началото на главата:

```
void ReadFile()
{
  OpenTheFile();
  while (FileHasMoreLines)
  {
    ReadNextLineFromTheFile();
    PrintTheLine();
  }
  CloseTheFile();
}
```

Нека сега преведем последователността от действия на български:

- Отваряме файл;
- Докато има следващ ред:

  - Четем следващ ред от файла;
  - Изписваме прочетения ред;
- Затваряме файла;

Методът е добре написан, но ако се вгледаме по-внимателно започват да възникват въпроси:

- Какво ще стане, ако няма такъв файл?
- Какво ще стане, ако файлът не може да се отвори (например, ако друг процес вече го е отворил за писане)?
- Какво ще стане, ако пропадне четенето на някой ред?
- Какво ще стане, ако файлът не може да се затвори?

Да допишем метода, така че да взима под внимание тези въпроси, без да използваме изключения, а да използваме кодове за грешка връщани от всеки използван метод. Кодовете за грешка са стандартен похват за обработка на грешките в процедурно ориентираното програмиране, при който всеки метод връща `int`, който дава информация дали методът е изпълнен правилно. Код за грешка 0 означава, че всичко е правилно, код различен от 0 означава някаква грешка. Различните видове грешки имат различен код (обикновено отрицателно число).

```
int ReadFile()
{
   errorCode = 0;
   openFileErrorCode = OpenTheFile();

   // Check whether the file is open
   if (openFileErrorCode == 0)
   {
      while (FileHasMoreLines)
      {
         readLineErrorCode = ReadNextLineFromTheFile();
         if (readLineErrorCode == 0)
         {
            // Line has been read properly
            PrintTheLine();
         }
         else
         {
            // Error during line reading
            errorCode = -1;
            break;
         }
      }
      closeFileErrorCode = CloseTheFile();
      if (closeFileErrorCode != 0 && errorCode == 0)
      {
        errorCode = -2;
      }
      else
      {
        errorCode = -3;
      }
    }
    else if (openFileErrorCode = -1)
    {
      // File does not exists
      errorCode = -4;
    }
    else if (openFileErrorCode = -2)
    {
      // File can’t be open
      errorCode = -5;
    }
    return errorCode;
  }
```

Както се вижда, се получава един доста замотан, трудно разбираем и лесно объркващ – "спагети" код. Логиката на програмата е силно смесена с логиката за обработка на грешките и непредвидените ситуации. По-голяма част от кода е тази за правилна обработка на грешките. Същинският код се губи сред обработката на грешки. Грешките нямат тип, нямат текстово описание (съобщение), нямат stack trace и трябва да гадаем какво означават кодовете -1, -2, -3 и т.н. Дори много хора биха се замислили как са програмирали програмистите на C и подобни езици едно време без изключения. Звучи толкова мазохистично като да чистиш леща с боксови ръкавици.

Всички тези нежелателни последици се избягват при използването на изключения. Ето колко по-прост и чист е псевдокодът на същия метод, само че с изключения:

```
void ReadFile()
{
  try
  {
    OpenTheFile();
    while (FileHasMoreLines)
    {
      ReadNextLineFromTheFile();
      PrintTheLine();
    }
  }
  catch (FileNotFoundException)
  {
    DoSomething();
  }
```

```
    catch (IOException)
    {
        DoSomethingElse();
    }
    finally
    {
        CloseTheFile();
    }
}
```

Всъщност изключенията не ни спестяват усилията при намиране и обработка на грешките, но ни позволяват да правим това по далеч по-елегантен, кратък, ясен и и ефективен начин.

Групиране на различните видове грешки

Йерархичната същност на изключенията позволява наведнъж да се прихващат и обработват цели групи изключения. Когато използваме **catch**, ние не прихващаме само дадения тип изключение, а цялата йерархия на типовете изключения, наследници на декларирания от нас тип.

```
catch (IOException e)
{
    // Handle IOException and all its descendants
}
```

Горният пример ще прихване не само **IOException**, но и всички негови наследници в това число **FileNotFoundException**, **EndOfStreamException**, **PathTooLongException** и много други. Няма да бъдат прихванати изключения като **UnauthorizedAccessException** (липса на права за извършване на дадена операция) **OutOfMemoryException** (препълване на паметта), тъй като те не са наследници на **IOException**. Ако се съмнявате кои изключения да прихванете, разгледайте йерархията на изключенията в MSDN.

Въпреки че не е добра практика, е възможно да направим прихващане на абсолютно всички изключения:

```
catch (Exception e)
{
    // A (too) general exception handler
}
```

Прихващането на **Exception** и всички негови наследници като цяло не е добра практика. За предпочитане е прихващането на по-конкретни групи от изключения като **IOException** или на един единствен тип изключение като например **FileNotFoundException**.

Предаване на грешките за обработка в стека на методите – прихващане на нива

Възможността за прихващането на изключения на нива е изключително удобна. Тя позволява обработката на изключението да се направи на най-подходящото място. Нека илюстрираме това с прост пример-сравнение с остарелия вече подход с връщане на кодове за грешка. Нека имаме следната структура от методи:

```
Method3()
{
  Method2();
}

Method2()
{
  Method1();
}

Method1()
{
  ReadFile();
}
```

Метода **Method3()** извиква **Method2()**, който от своя страна извиква **Method1()** където се вика **ReadFile()**. Да предположим, че **Method3()** е този, който се интересува от възможна възникнала грешка в метода **ReadFile()**. Ако възникне такава грешка в **ReadFile()**, при традиционния подход с кодове на грешка прехвърлянето й до **Method3()** не би било никак лесно:

```
void Method3()
{
  errorCode = Method2();
  if (errorCode != 0)
    process the error;
  else
    DoTheActualWork();
}

int Method2()
{
  errorCode = Method1();
  if (errorCode != 0)
    return errorCode;
  else
    DoTheActualWork();
}
```

```
int Method1()
{
   errorCode = ReadFile();
   if (errorCode != 0)
      return errorCode;
   else
      DoTheActualWork();
}
```

Като начало в **`Method1()`** трябва анализираме кода за грешка връщан от метода **`ReadFile()`** и евентуално да предадем на **`Method2()`**. В **`Method2()`** трябва да анализираме кода за грешка връщан от **`Method1()`** и евентуално да го предадем на **`Method3()`**, където да се обработи самата грешка.

Как можем да избегнем всичко това? Да си припомним, че средата за изпълнение (CLR) търси прихващане на изключения назад в стека на извикване на методите и позволява на всеки един от методите в стека да дефинира прихващане и обработка на изключенията. Ако методът не е заинтересован да прихване някое изключение, то просто се препраща назад в стека:

```
void Method3()
{
   try
   {
      Method2();
   }
   catch (Exception e)
   {
      process the exception;
   }
}

void Method2()
{
   Method1();
}

void Method1()
{
   ReadFile();
}
```

Ако възникне грешка при четенето на файла, то тя ще се пропусне от **`Method1()`** и **`Method2()`** и ще се прихване и обработи чак в **`Method3()`**, където всъщност е най-подходящото място за обработка на грешката. Да си припомним отново най-важното правило: всеки метод трябва да прихваща само грешките, които е компетентен да обработи и трябва да пропуска всички останали грешки.

Добри практики при работа с изключения

В настоящата секция ще дадем някои препоръки и утвърдени практики за правилно използване на механизмите на изключенията за обработка на грешки и необичайни ситуации. Това са важни правила, които трябва да запомните и следвате. Не ги пренебрегвайте!

Кога да разчитаме на изключения?

За да разберем кога е добре да разчитаме на изключения и кога не, нека разгледаме следния пример: имаме програма, която отваря файл по зададени път и име на файл. Потребителят може да обърка името на файла докато го пише. Тогава това събитие по-скоро трябва да се счита за нормално, а не за изключително.

Срещу подобно събитие можем да се защитим като първо проверим дали файлът съществува и чак тогава да се опитаме да го отворим:

```
static void ReadFile(string fileName)
{
   if (!File.Exists(fileName))
   {
      Console.WriteLine(
         "The file '{0}' does not exist.", fileName);
      return;
   }

   StreamReader reader = new StreamReader(fileName);
   using (reader)
   {
      while (!reader.EndOfStream)
      {
         string line = reader.ReadLine();
         Console.WriteLine(line);
      }
   }
}
```

Ако изпълним метода и файлът липсва, ще получим следното съобщение на конзолата:

```
The file 'WrongTextFile.txt' does not exist.
```

Другият вариант да имплементираме същата логика е следният:

```
static void ReadFile(string filename)
{
   StreamReader reader = null;
   try
   {
        reader = new StreamReader(filename);
        while (!reader.EndOfStream)
        {
            string line = reader.ReadLine();
            Console.WriteLine(line);
        }
        reader.Close();
    }
    catch (FileNotFoundException)
    {
        Console.WriteLine(
            "The file '{0}' does not exist.", filename);
    }
    finally
    {
        if (reader != null)
        {
            reader.Close();
        }
    }
}
```

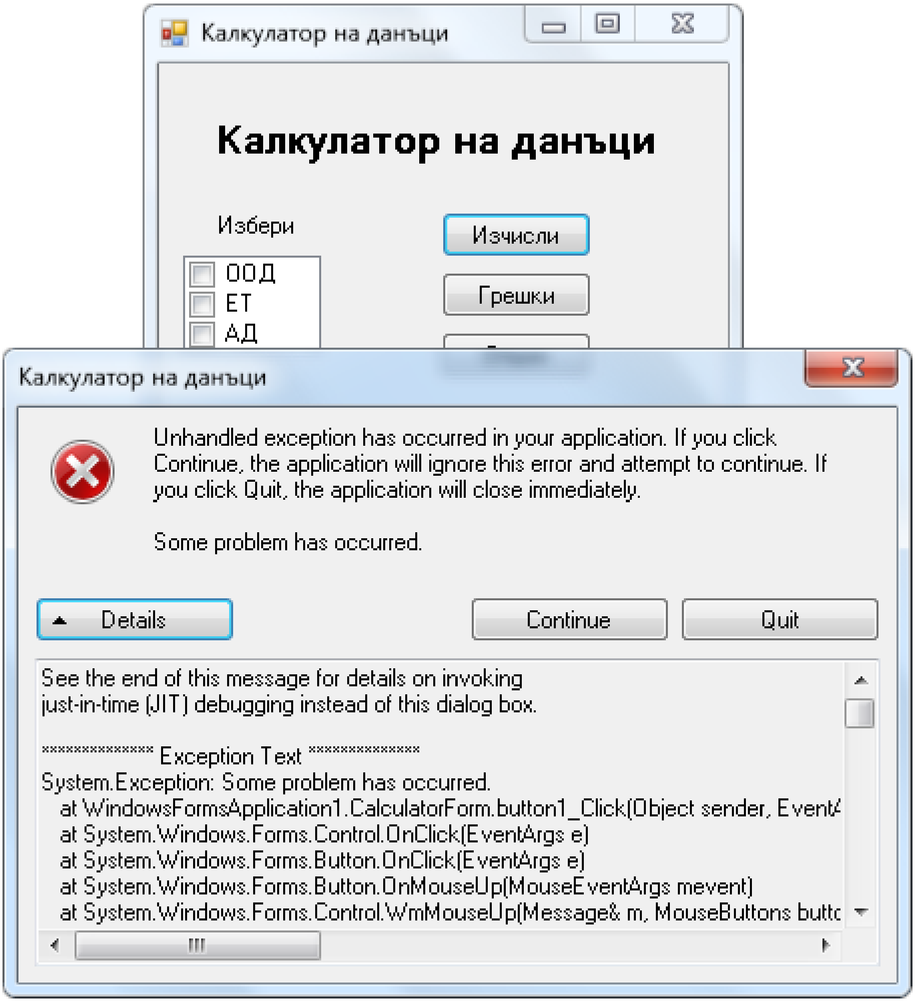
По принцип вторият вариант се счита за по-лош, тъй като изключенията трябва да се ползват за изключителни ситуации, а липсата на файла в нашия случай е по-скоро обичайна ситуация.

Недобра практика е да се разчита на изключения за обработка на очаквани събития и от още една гледна точка: производителност. Хвърлянето на изключение е бавна операция, защото трябва да се създаден обект, съдържащ изключението, да се инициализира stack trace, да се открие обработчик на това изключение и т.н.

Точната граница между очаквано и неочаквано поведение е трудно да бъде ясно дефинирана. Най-общо очаквано събитие е нещо свързано с функционалността на програмата. Въвеждането на грешно име на файла е пример за такова. Спирането на тока докато работи програмата, обаче не е очаквано събитие.

Да хвърляме ли изключения на потребителя?

Изключенията са неясни и объркващи за обикновения потребител. Те създават впечатление за лошо написана програма, която "гърми неконтролирано" и "има бъгове". Представете си какво ще си помисли една възрастна служителка (с оглед на дадения пример), която въвежда фактури, ако внезапно приложението й покаже следния диалог:


Калкулатор на данъци
Калкулатор на данъци
Избери
☐ ООД
☐ ЕТ
☐ АД
Изчисли
Грешки
Калкулатор на данъци
Details
Continue
Quit

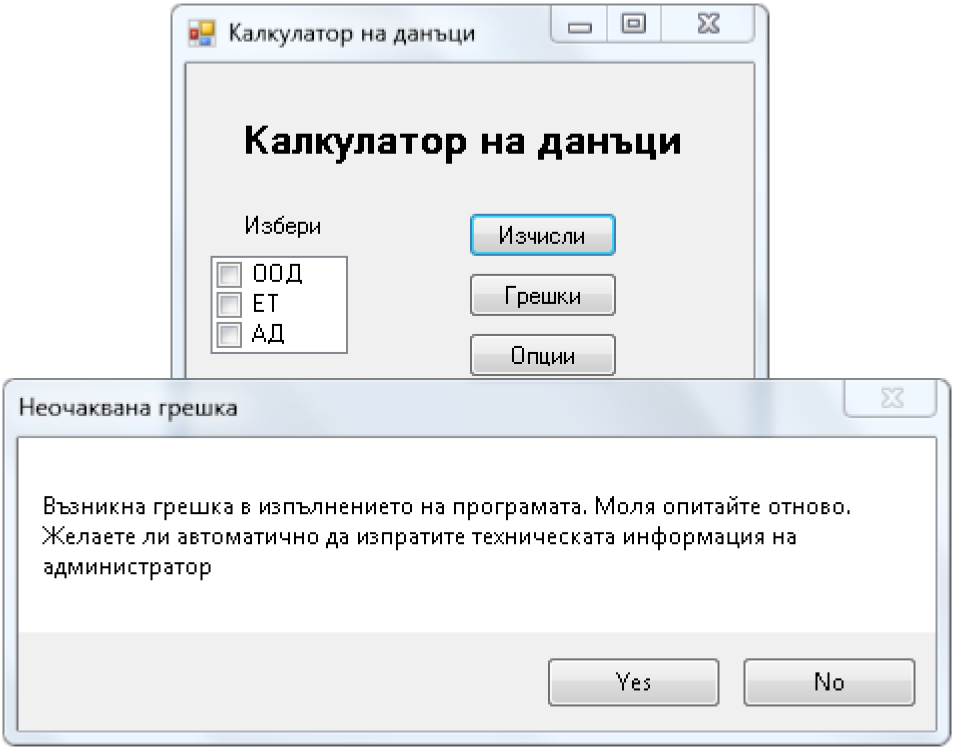
Unhandled exception has occurred in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will close immediately.

Some problem has occurred.

```
See the end of this message for details on invoking
just-in-time (JIT) debugging instead of this dialog box.

************** Exception Text **************
System.Exception: Some problem has occurred.
   at WindowsFormsApplication1.CalculatorForm.button1_Click(Object sender, Event/
   at System.Windows.Forms.Control.OnClick(EventArgs e)
   at System.Windows.Forms.Button.OnClick(EventArgs e)
   at System.Windows.Forms.Button.OnMouseUp(MouseEventArgs mevent)
   at System.Windows.Forms.Control.WmMouseUp(Message& m, MouseButtons buttc
```

Този диалог е много подходящ за технически лица (например програмисти и администратори), но е изключително неподходящ за крайния потребител (особено, когато той няма технически познания).

Вместо този диалог можем да покажем друг, много по-дружелюбен и разбираем за обикновения потребител диалог:



Възникна грешка в изпълнението на програмата. Моля опитайте отново. Желаете ли автоматично да изпратите техническата информация на администратор

Това е добрият начин да показваме съобщения за грешка: хем да има разбираемо съобщение на езика на потребителя (в случая на български език), хем да има и техническа информация, която може да бъде извлечена при нужда, но не се показва в самото начало, за да не стряска потребителите.

Препоръчително е изключения, които не са хванати от никой (такива може да са само runtime изключенията), да се хващат от общ глобален "прихващач", който да ги записва (в най-общия случай) някъде по твърдия диск, а на потребителя да показва "приятелско" съобщение в стил: "Възникна грешка, опитайте по-късно". Добре е винаги да показвате освен съобщение разбираемо за потребителя и техническа информация (stack trace), която обаче да е достъпна само ако потребителят я поиска.

Хвърляйте изключенията на съответното ниво на абстракция!

Когато хвърляте ваши изключения, съобразявайте се с абстракциите, в контекста, на които работи вашият метод. Например, ако вашият метод се отнася за работа с масиви, може да хвърлите **`IndexOutOfRangeException`** или **`NullReferenceException`**, тъй като вашият метод работи на ниско ниво и оперира директно с паметта и с елементите на масивите. Ако, обаче имате метод, който извършва олихвяване на всички сметки в една банка, той не трябва да хвърля **`IndexOutOfRangeException`**, тъй като това изключение не е от бизнес областта на банковия сектор и олихвяването.

Нормално е олихвяването в банковия софтуер да хвърли изключение `InvalidInterestException` с подходящо съобщение за грешка от бизнес областта на банките, за което би могло да бъде закачено (вложено) оригиналното изключение `IndexOutOfRangeException`.

Представете си да сте си купили билет за автобус и пристигайки на автогарата омаслен монтьор да ви обясни, че ходовата част на автобуса има нужда от регулиране и кормилната рейка е нестабилна. Освен, ако не сте монтьор или специалист по автомобили, тази информация не ви помага с нищо. Нито става ясно колко ще се забави вашето пътуване, нито дали въобще ще пътувате. Вие очаквате, ако има проблем, да ви посрещне усмихната девойка от фирмата-превозвач и да ви обясни, че резервният автобус ще дойде след 10 минути и до тогава можете да изчакате на топло в кафенето.

Същото е при програмирането – ако хвърляте изключения, които не са от бизнес областта на компонента или класа, който разработвате, има голям шанс да не ви разберат и грешката да не бъде обработена правилно.

Можем да дадем още един пример: извикваме метод, който сортира масив с числа и той хвърля изключение `TransactionAbortedException`. Това е също толкова неадекватно съобщение, колкото и `NullReferenceException` при изпълнение на олихвяването в една банка. Веднага ще си помислите "Каква транзакция, какви пет лева? Нали сортираме масив!" и този въпрос е напълно адекватен. Затова се съобразявайте с нивото на абстракция, на което работи даденият метод, когато хвърляте изключение от него.

Ако изключението има причинител, запазвайте го!

Винаги, когато при прихващане на изключение хвърляте ново изключение от по-високо ниво на абстракция, закачайте за него оригиналното изключение. По този начин ползвателите на вашия код ще могат по-лесно да установят точната причина за грешката и точното място, където тя възниква в началния момент.

Това правило е частен случай на по-генералното правило:

Всяко изключение трябва да носи в себе си максимално подробна информация за настъпилия проблем.

От горното правило произтичат много други по-конкретни правила, например, че съобщението за грешка трябва да е адекватно, че типът на съобщението трябва да е адекватен на възникналия проблем, че изключението трябва да запазва причинителя си и т.н.

Давайте подробно описателно съобщение при хвърляне на изключение!

Съобщението за грешка, което всяко изключение носи в себе си е изключително важно. В повечето случаи то е напълно достатъчно, за да разберете какъв точно е проблемът, който е възникнал. Ако съобщението е неадекватно, ползвателите на вашия метод няма да са щастливи и няма да решат бързо проблема.

Да вземем един пример: имате метод, който прочита настройките на дадено приложение от текстов файл. Това са примерно местоположенията и размерите на всички прозорци в приложението и други настройки. Случва се проблем при четенето на файла с настройките и получавате съобщение за грешка:

```
Error.
```

Това достатъчно ли ви е, за да разберете какъв е проблемът? Очевидно не, нали? Какво съобщение трябва да дадем, така че то да е достатъчно информативно? Това съобщение по-добро ли е?

```
Error reading settings file.
```

Очевидно горното съобщение е по-адекватно, но е все още недостатъчно. То обяснява каква е грешката, но не обяснява причината за възникването ѝ. Да предположим, че променим програмата, така че да дава следната информация за грешката:

```
Error reading settings file:
C:\Users\Administrator\MyApp\MyApp.settings
```

Това съобщение очевидно е по-добро, защото ни подсказва в кой файл е проблемът (нещо, което би ни спестило много време, особено ако не сме запознати с приложението и не знаем къде точно то пази файла с настройките си). Може ситуацията да е дори по-лоша – може да нямаме сорс кода на въпросното приложение или модул, който генерира грешката. Тогава е възможно да нямаме пълен stack trace (ако сме компилирали без дебъг информация) или ако имаме stack trace, той не ни върши работа, защото нямаме сорс кода на проблемния файл, хвърлил изключението. Затова съобщението за грешка трябва да е още по-подробно, например като това:

```
Error reading settings file:
C:\Users\Administrator\MyApp\MyApp.settings. Number expected at
line 17.
```

Това съобщение вече само говори за проблема. Очевидно имаме грешка на ред 17 във файла **`MyApp.settings`**, който се намира в папката **`C:\Users\Administrator\MyApp`**. В този ред трябва да има число, а има нещо друго. Ако отворим файл, бързо можем да намерим проблема, нали?

Изводът от този пример е само един:

Винаги давайте адекватно, подробно и конкретно съобщение за грешка, когато хвърляте изключение! Ползвателят на вашия код трябва само като прочете съобщението, веднага да му стане ясно какъв точно е проблемът, къде се е случил и каква е причината за него.

Ще дадем още няколко примера:

- Имаме метод, който търси число в масив. Ако той хвърли **IndexOutOfRangeException**, от изключително значение е в съобщението за грешка да се упомене индексът, който не може да бъде достъпен, примерно 18 при масив с дължина 7. Ако не знаем позицията, трудно ще разберем защо се получава излизане от масива.
- Имаме метод, който чете числа от файл. Ако във файла се срещне някой ред, на който няма число, би трябвало да получим грешка, която обяснява, че на ред 17 (примерно) се очаква число, а там има символен низ (и да се отпечата точно какъв символен низ има там).
- Имаме метод, който изчислява стойността на числен израз. Ако намерим грешка в израза, изключението трябва да съобщава каква грешка е възникнала и на коя позиция. Кодът, който предизвиква грешката може да ползва **String.Format(…)**, за да построи съобщението за грешка. Ето един пример:

```
throw new FormatException(
   string.Format("Invalid character at position {0}. " +
   "Number expected but character '{1}' found.", index, ch));
```

Съобщение за грешка с невярно съдържание

Има само едно нещо по-лошо от изключение без достатъчно информация и то е изключение с грешна информация. Например, ако в последния пример съобщим за грешка на ред 3, а грешката е на ред 17, това е изключително заблуждаващо и е по-вредно, отколкото просто да кажем, че има грешка без подробности.

Внимавайте да не отпечатвате съобщения за грешка с невярно съдържание!

За съобщенията за грешки използвайте английски

Използвайте английски език в съобщенията за грешки. Това правило е много просто. То е частен случай на принципа, че целият сорс код на програмите ви (включително коментарите и съобщенията за грешки) трябва да са на английски език. Причината за това е, че английският е единственият език, който е разбираем за всички програмисти по света.

Никога не знаете дали кодът, който пишете няма в някой слънчев ден да се ползва от чужденци. Хубаво ли ще ви е, ако ползвате чужд код и той ви съобщава за грешки примерно на виетнамски език?

Никога не игнорирайте прихванатите изключения!

Никога не игнорирайте изключенията, които прихващате, без да ги обработите. Ето един пример как не трябва да правите:

```
try
{
   string fileName = "WrongTextFile.txt";
   ReadFile(fileName);
}
catch (Exception e)
{ }
```

В този пример авторът на този ужасен код прихваща изключенията и ги игнорира. Това означава, че ако липсва файлът, който търсим, програмата няма да прочете нищо от него, но няма и да съобщи за грешка, а ползвателят на този код ще бъде заблуден, че файлът е бил прочетен, като всъщност той липсва.

Начинаещите програмисти понякога пишат такъв код. Вие нямате причина да пишете такъв код, нали?

Ако понякога се наложи да игнорирате изключение, нарочно и съзнателно, добавете изричен коментар, който да помага при четене на кода. Ето един пример:

```
int number = 0;
try
{
   string line = Console.ReadLine();
   number = Int32.Parse(line);
}
catch (Exception)
{
   // Incorrect numbers are intentionally considered 0
}
Console.WriteLine("The number is: " + number);
```

Кодът по-горе може да се подобри като или се използва **`Int32.TryParse(…)`** или като променливата **`number`** се занулява в **`catch`** блока, а не предварително. Във втория случай коментарът в кода няма да е необходим и няма да има нужда от празен **`catch`** блок.

Отпечатвайте съобщенията за грешка на конзолата само в краен случай!

Представете си например нашия метод, който чете настройките на приложението от текстов файл. Ако възникне грешка, той би могъл да я отпечата на конзолата, но какво ще стане с извикващия метод? Той ще си помисли, че настройките са били успешно прочетени, нали?

Има едно много важно правило в програмирането:

Един метод или трябва да върши работата, за която е предназначен, или трябва да хвърля изключение.

Това правило е много, много важно и затова ще го повторим в малко по-разширена форма:

Един метод или трябва да върши работата, за която е предназначен, или трябва да хвърля изключение. При грешни входни данни методът трябва да връща изключение, а не грешен резултат!

Това правило можем да обясним в по-големи детайли: Един метод се пише, за да свърши някаква работа. Какво върши методът трябва да става ясно от неговото име. Ако не можем да дадем добро име на метода, значи той прави много неща и трябва да се раздели на части, всяка от които да е в отделен метод. Ако един метод не може да свърши работата, за която е предназначен, той трябва да хвърли изключение. Например, ако имаме метод за сортиране на масив с числа, ако масивът е празен, методът или трябва да върне празен масив, или да съобщи за грешка. Грешните входни данни трябва да предизвикват изключение, не грешен резултат! Например, ако се опитаме да вземем от даден символен низ с дължина 10 символа подниз от позиция 7 до позиция 12, трябва да получим изключение, не да върнем по-малко символи. Ако обърнете внимание, ще се уверите, че точно така работи методът **`Substring()`** в класа **`String`**.

Ще дадем още един, по-убедителен пример, който потвърждава правилото, че един метод или трябва да свърши работата, за която е написан, или трябва да хвърли изключение. Да си представим, че копираме голям файл от локалния диск към USB flash устройство. Може да се случи така, че мястото на flash устройството не достига и файлът не може да бъде копиран. Кое от следните е правилно да направи програмата за копиране на файлове (примерно Windows Explorer)?

- Файлът не се копира и копирането завършва тихо, без съобщение за грешка.
- Файлът се копира частично, доколкото има място на flash устройството. Част от файла се копира, а останалата част се отрязва. Не се показва съобщение за грешка.

- Файлът се копира частично, доколкото има място на flash устройството и се показва съобщение за грешка.
- Файлът не се копира и се показва съобщение за грешка.

Единственото коректно от гледна точка на очакванията на потребителя поведение е последното: при проблем файлът трябва да не се копира частично и трябва да се покаже съобщение за грешка. Същото важи, ако трябва да напишем метод, който копира файлове. Той или трябва да копира напълно и до край зададения файл или трябва да предизвика изключение като същевременно не оставя следи от започната и недовършена работа (т.е. трябва да изтрие частичният резултат, ако е създаден такъв).

Не прихващайте всички изключения!

Една много често срещана грешка при работата с изключения е да се прихващат всички грешки, без оглед на техния тип. Ето един пример, при който грешките се обработват некоректно:

```
try
{
  ReadFile("CorrectTextFile.txt");
}
catch (Exception)
{
  Console.WriteLine("File not found.");
}
```

В този код предполагаме, че имаме метод **`ReadFile()`**, който прочита текстов файл и го връща като **`string`**. Забелязваме, че **`catch`** блокът прихваща наведнъж всички изключения (независимо от типа им), не само **`FileNotFoundException`**, и при всички случаи отпечатва, че файлът не е намерен. Хубаво, обаче има ситуации, които са непредвидени. Например какво става, когато файлът е заключен от друг процес в операционната система. В такъв случай средата за изпълнение CLR ще генерира **`UnauthorizedAccessException`**, но съобщението за грешка, което програмата ще изведе към потребителя, ще е грешно и подвеждащо. Файлът ще го има, а програмата ще твърди, че го няма, нали? По същия начин, ако при отварянето на файла свърши паметта, ще се генерира съобщение **`OurOfMemoryException`**, но отпечатаната грешка ще е отново некоректна.

Прихващайте само изключения, от които разбирате и знаете как да обработите!

Какъв е изводът от последния пример? Трябва да обработваме само грешките, които очакваме и за които сме подготвени. Останалите грешки (изключения) не трябва въобще да ги прихващаме, а трябва да ги оставим да ги прихване някой друг метод, който знае какво да ги прави.

Един метод трябва да прихваща само изключенията, които е компетентен да обработи адекватно, а не всички.

Това е много важно правило, което непременно трябва да спазвате. Ако не знаете как да обработите дадено изключение, или не го прихващайте, или го обгърнете с ваше изключение и го хвърлете по стека да си намери обработчик.

Упражнения

1. Да се намерят всички стандартни изключения от йерархията на **`System.IO.IOException`**.
2. Да се намерят всички стандартни изключения от йерархията на **`System.IO.FileNotFoundException`**.
3. Да се намерят всички стандартни изключения от йерархията на **`System.ApplicationException`**.
4. Обяснете какво представляват изключенията, кога се използват и как се прихващат.
5. Обяснете ситуациите, при които се използва **`try-finally`** конструкцията. Обяснете връзката между **`try-finally`** и **`using`** конструкциите.
6. Обяснете предимствата на използването на изключения.
7. Напишете програма, която прочита от конзолата цяло положително число и отпечатва на конзолата корен квадратен от това число. Ако числото е отрицателно или невалидно, да се изпише "Invalid Number" на конзолата. Във всички случаи да се принтира на конзолата "Good Bye".
8. Напишете метод **`ReadNumber(int start, int end)`**, който въвежда от конзолата число в диапазона [**`start…end`**]. В случай на въведено невалидно число или число, което не е в подадения диапазон хвърлете подходящо изключение. Използвайки този метод напишете програма, която въвежда 10 числа a_1, a_2, …, a_{10}, такива, че **`1 < a1 < … < a10 < 100`**.
9. Напишете метод, който приема като параметър име на текстов файл, прочита съдържанието му и го връща като **`string`**. Какво е правилно да направи методът с евентуално възникващите изключения?
10. Напишете метод, който приема като параметър име на бинарен файл и прочита съдържанието на файла и го връща като масив от байтове. Напишете метод, който записва прочетеното съдържание в друг файл. Сравнете двата файла.
11. Потърсете информация в Интернет и дефинирайте собствен клас за изключение **`FileParseException`**. Вашето изключение трябва да съдържа в себе си името на файл, който се обработва и номер на ред, в

който е възникнал проблем. Добавете подходящи конструктори за вашето изключение. Напишете програма, която чете от текстов файл числа. Ако при четенето се стигне до ред, който не съдържа число, хвърлете **`FileParseException`** и го обработете в извикващия метод.

12. Напишете програма, която прочита от потребителя пълен път до даден файл (например C:\Windows\win.ini), прочита съдържанието на файла и го извежда на конзолата. Намерете в MSDN как да използвате метода **`System.IO.File.ReadAllText(…)`**. Уверете се, че прихващате всички възможни изключения, които могат да възникнат по време на работа на метода и извеждайте на конзолата съобщения за грешка, разбираеми за обикновения потребител.
13. Напишете програма, която изтегля файл от Интернет по даден URL адрес, примерно (http://www.devbg.org/img/Logo-BASD.jpg).

Решения и упътвания

1. Потърсете в MSDN. Най-лесният начин да направите това е да напишете в Google "IOException MSDN".
2. Разгледайте упътването за предходната задача.
3. Разгледайте упътването за предходната задача.
4. Използвайте информацията от началото на настоящата тема.
5. При затруднения използвайте информацията от секцията "Конструкцията try-finally".
6. При затруднения използвайте информацията от секцията "Предимства при използване на изключения".
7. Направете **`try{}`** - **`catch(){}`** - **`finally{}`** конструкция.
8. При въведено невалидно число може да хвърляте изключението Exception поради липсва на друг клас изключения, който по-точно да описва проблема. Алтернативно можете да дефинирате собствен клас изключение **`InvalidNumberException`**.
9. Първо прочетете главата "Текстови файлове". Прочетете файла ред по ред с класа **`System.IO.StreamReader`** и добавяйте редовете в **`System.Text.StringBuilder`**. Изхвърляйте всички изключения от метода без да ги прихващате.
10. Малко е вероятно да напишете коректно този метод от първи път без чужда помощ. Първо прочетете в Интернет как се работи с бинарни потоци. След това следвайте препоръките по-долу за четенето на файла:
    - Използвайте за четене **`FileStream`**, а прочетените данни записвайте в **`MemoryStream`**. Трябва да четете файла на части, примерно на последователни порции по 64 KB, като последната порция може да е по-малка.

- Внимавайте с метода за четене на байтове `FileStream.Read( byte[] buffer, int offset, int count)`. Този метод може да прочете по-малко байтове, отколкото сте заявили. Колкото байта прочетете от входния поток, толкова трябва да запишете. Трябва да организирате цикъл, който завършва при връщане на стойност 0 за броя прочетени байтове.
- Използвайте `using`, за да затваряте коректно потоците.

Записването на масив от байтове във файл е далеч по-проста задача. Отворете `FileStream` и започнете да пишете в него байтовете от `MemoryStream`. Използвайте `using`, за да затваряте потоците правилно.

Накрая тествайте с някой много голям ZIP архив (примерно 300 MB). Ако програмата ви работи некоректно, ще счупвате структурата на архива и ще се получава грешка при отварянето му.

11. Наследете класа `Exception` и добавете подходящ конструктор, примерно `FileParseException(string message, string filename, int line)`. След това ползвайте вашето изключение както ползвате за всички други изключения, които познавате. Числата можете да четете с класа `StreamReader`.

12. Потърсете всички възможни изключения, които възникват в следствие на работата на метода и за всяко от тях дефинирайте `catch` блок.

13. Потърсете в Интернет статии на тема изтегляне на файл от C#. Ако се затруднявате, потърсете информация и примери за използване конкретно на класа `WebClient`. Уверете се, че прихващате и обработвате правилно всички изключения, които могат да възникнат.

Глава 13. Символни низове

В тази тема...

В настоящата тема ще се запознаем със символните низове. Ще обясним как те са реализирани в C# и по какъв начин можем да обработваме текстово съдържание. Ще прегледаме различни методи за манипулация на текст: ще научим как да сравняваме низове, как да търсим поднизове, как да извличаме поднизове по зададени параметри, както и да разделяме един низ по разделители. Ще предоставим кратка, но много полезна информация за регулярните изрази. Ще разгледаме някои класове за ефективно построяване на символни низове. Накрая ще се запознаем с методи и класове за по-елегантно и стриктно форматиране на текстовото съдържание.

Символни низове

В практиката често се налага обработката на текст: четене на текстови файлове, търсене на ключови думи и заместването им в даден параграф, валидиране на входни потребителски данни и др. В такива случаи можем да запишем текстовото съдържание, с което ще боравим, в символни низове, и да го обработим с помощта на езика C#.

Какво е символен низ (стринг)?

Символният низ е последователност от символи, записана на даден адрес в паметта. Помните ли типа **`char`**? В променливите от тип **`char`** можем да запишем само един символ. Когато е необходимо да обработваме повече от един символ на помощ идват низовете.

В .NET Framework всеки символ има пореден номер от **Unicode** таблицата. Стандартът Unicode е създаден в края на 80-те и началото на 90-те години с цел съхраняването на различни типове текстови данни. Предшественикът му ASCII позволява записването на едва 128 или 256 символа (съответно ASCII стандарт със 7-битова или 8-битова таблица). За съжаление, това често не удовлетворява нуждите на потребителя – тъй като в 128 символа могат да се поберат само цифри, малки и главни латински букви и някои специални знаци. Когато се наложи работа с текст на кирилица или друг специфичен език (например азиатски или африкански), 128 или 256 символа са крайно недостатъчни. Ето защо .NET използва 16-битова кодова таблица за символите. С помощта на знанията ни за бройните системи и представянето на информацията в компютрите, можем да сметнем, че кодовата таблица съхранява 2^16 = 65536 символа. Някои от символите се кодират по специфичен начин, така че е възможно използването на два символа от Unicode таблицата за създаване на нов символ – така получените знаци надхвърлят 100 000.

Класът System.String

Класът **`System.String`** позволява обработка на символни низове в C#. За декларация на низовете ще продължим да използваме служебната дума **`string`**, която е **псевдоним** (**alias**) в C# на класа **`System.String`** от .NET Framework. Работата със **`string`** ни улеснява при манипулацията на текстови данни: построяване на текстове, търсене в текст и много други операции. Пример за декларация на символен низ:

```
string greeting = "Hello, C#";
```

Декларирахме променливата **`greeting`** от тип **`string`**, която има съдържание "**`Hello, C#`**". Представянето на съдържанието в символния низ изглежда по подобен начин:

H	**e**	**l**	**l**	**o**	**,**		**C**	**#**

Вътрешното представяне на класа е съвсем просто – масив от символи. Можем да избегнем използването на класа, като декларираме променлива от тип **`char[]`** и запълним елементите на масива символ по символ. Недостатъците на това обаче са няколко:

1. Запълването на масива става символ по символ, а не наведнъж.
2. Трябва да знаем колко дълъг ще е текстът, за да сме наясно дали ще се побере в заделеното място за масива.
3. Обработката на текстовото съдържание става ръчно.

Класът String – универсално решение?

Използването на **`System.String`** не е идеално и универсално решение – понякога е уместно използването на други символни структури.

В C# съществуват и други класове за обработка на текст – с някои от тях ще се запознаем по-нататък в настоящата тема.

Типът **`string`** е по-особен от останалите типове данни. Той е клас и като такъв той спазва принципите на обектно-ориентираното програмиране. Стойностите му се записват в динамичната памет, а променливите от тип **`string`** пазят препратка към паметта (референция към обект в динамичната памет).

Класът **`string`** има важна особеност – последователностите от символи, записани в променлива от класа, са **неизменими (immutable)**. След като е веднъж зададено, съдържанието на променливата не се променя директно – ако опитаме да променим стойността, тя ще бъде записана на ново място в динамичната памет, а променливата ще започне да сочи към него. Тъй като това е важна особеност, тя ще бъде онагледена малко по-късно.

Стринговете и масиви от символи

Стринговете много приличат на масиви от символи (**`char[]`**), но за разлика от тях не могат да се променят. Подобно на масивите те имат свойство **`Length`**, което връща дължината на низа, и позволяват достъп по индекс. Индексирането, както и при масивите, става по индекси от **`0`** до **`Length-1`**. Достъпът до символа на дадена позиция в даден стринг става с оператора **`[]`** (индексатор), но е позволен само за четене:

```
string str = "abcde";
char ch = str[1]; // ch == 'b'
str[1] = 'a'; // Compilation error
ch = str[50]; // IndexOutOfRangeException
```

Символни низове – прост пример

Да дадем един пример за използване на променливи от тип **`string`**:

```
string message = "Stand up, stand up, Balkan superman.";

Console.WriteLine("message = {0}", message);
Console.WriteLine("message.Length = {0}", message.Length);

for (int i = 0; i < message.Length; i++)
{
  Console.WriteLine("message[{0}] = {1}", i, message[i]);
}
// Console output:
// message = "Stand up, stand up, Balkan superman."
// message.Length = 36
// message[0] = S
// message[1] = t
// message[2] = a
// message[3] = n
// message[4] = d
// ...
```

Обърнете внимание на стойността на стринга – кавичките не са част от текста, а ограждат стойността му. В примера е демонстрирано как може да се отпечатва символен низ, как може да се извлича дължината му и как може да се извличат символите, от които е съставен.

Escaping при символните низове

Както вече знаем, ако искаме да използваме кавички в съдържанието на символен низ, трябва да поставим наклонена черта преди тях за да укажем, че имаме предвид самия символ кавички, а не използване кавичките за начало / край на низ:

```
string quote = "Book’s title is \"Intro to C#\"";
// Book's title is "Intro to C#"
```

Кавичките в примера са част от текста. В променливата те са добавени чрез поставянето им след екраниращия знак обратна наклонена черта (\). По този начин компилаторът разбира, че кавичките не служат за начало или край на символен низ, а са част от данните. Избягването на специалните символи в сорс кода се нарича екраниране (escaping).

Деклариране на символен низ

Можем да декларираме променливи от тип символен низ по следния начин:

```
string str;
```

Декларацията на символен низ представлява декларация на променлива от тип **`string`**. Това не е еквивалентно на създаването на променлива и заделянето на памет за нея! С декларацията уведомяваме компилатора, че ще използваме променлива **`str`** и очакваният тип за нея е **`string`**. Ние не създаваме променливата в паметта и тя все още не е достъпна за обработки (има стойност **`null`**, което означава липса на стойност).

Създаване и инициализиране на символен низ

За да може да обработваме декларираната стрингова променлива, трябва да я създадем и инициализираме. Създаването на променлива на клас (познато още като **инстанциране**) е процес, свързан със заделянето на област от динамичната памет. Преди да зададем конкретна стойност на символния низ, стойността му е **`null`**. Това може да бъде объркващо за начинаещия програмист: неинициализираните променливи от типа **`string`** не съдържат празни стойности, а специалната стойност **`null`** – и опитът за манипулация на такъв стринг ще генерира грешка (изключение за достъп до липсваща стойност **`NullReferenceException`**)!

Можем да инициализираме променливи по 3 начина:

1. Чрез задаване на низов литерал.
2. Чрез присвояване стойността от друг символен низ.
3. Чрез предаване стойността на операция, връщаща символен низ.

Задаване на литерал за символен низ

Задаването на литерал за символен низ означава предаване на предефинирано текстово съдържание на променлива от тип **`string`**. Използваме такъв тип инициализация, когато знаем стойността, която трябва да се съхрани в променливата. Пример за задаване на литерал за символен низ:

```
string website = "http://www.introprogramming.info/";
```

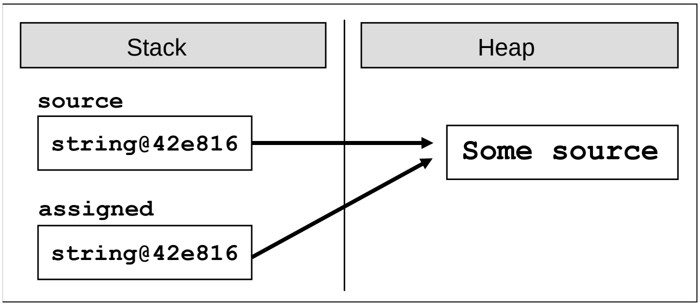
В този пример създаваме променливата **`website`** и й задаваме като стойност символен литерал.

Присвояване стойността на друг символен низ

Присвояването на стойността е еквивалентно на насочване на **`string`** стойност или променлива към дадена променлива от тип **`string`**. Пример за това е следният фрагмент:

```
string source = "Some source";
string assigned = source;
```

Първо, декларираме и инициализираме променливата **`source`**. След това променливата **`assigned`** приема стойността на **`source`**. Тъй като класът **`string`** е референтен тип, текстът "**`Some source`**" е записан в динамичната памет (heap, хийп) на място, сочено от първата променлива.



На втория ред пренасочваме променливата **`assigned`** към същото място, към което сочи другата променлива. Така двата обекта получават един и същ адрес в динамичната памет и следователно една и съща стойност.

Промяната на коя да е от променливите обаче ще се отрази **само и единствено** на нея, поради неизменимостта на типа **`string`**, тъй като при промяната ще се създаде копие на променяния стринг. Това не се отнася за останалите референтни типове (нормалните изменими типове), защото при тях промените се нанасят директно на адреса в паметта и всички референции сочат към променения обект.

Предаване стойността на операция, връщаща символен низ

Третият вариант за инициализиране на символен низ е предаването на стойността на израз или операция, която връща стрингов резултат. Това може да бъде резултат от метод, който валидира данни; събиране на стойностите на няколко константи и променливи, преобразуване на съществуваща променлива и др. Пример за израз, връщащ символен низ:

```csharp
string email = "some@gmail.com";
string info = "My mail is: " + email;
// My mail is: some@gmail.com
```

Променливата **`info`** е създадена от **съединяването** (**concatenation**) на литерали и променлива.

Четене и печатане на конзолата

Нека сега разгледаме как можем да четем символни низове, въведени от потребителя, и как можем да печатаме символни низове на стандартния изход (на конзолата).

Четене на символни низове

Четенето на символни низове може да бъде осъществено чрез методите на познатия ни клас **System.Console**:

```
string name = Console.ReadLine();
```

В примера прочитаме от конзолата входните данни чрез метода **ReadLine()**. Той предизвиква потребителя да въведе стойност и да натисне **[Enter]**. След натискане на клавиша **[Enter]** променливата **name** ще съдържа въведеното име от клавиатурата.

Какво можем да правим, след като променливата е създадена и в нея има стойност? Можем например да я използваме в изрази с други символни низове, да я подаваме като параметър на методи, да я записваме в текстови документи и др. На първо време, можем да я изведем на конзолата, за да се уверим, че данните са прочетени коректно.

Отпечатване на символни низове

Извеждането на данни на стандартния изход се извършва също чрез познатия ни клас **System.Console**:

```
Console.WriteLine("Your name is: " + name);
```

Използвайки метода **WriteLine(…)** извеждаме съобщението: "**Your name is:**", следвано от стойността на променливата **name**. След края на съобщението се добавя символ за нов ред. Ако искаме да избегнем символа за нов ред и съобщенията да се извеждат на един и същ, тогава прибягваме към метода **Write(…)**.

Можем да си припомним класа **System.Console** от темата "Вход и изход от конзолата".

Операции върху символни низове

След като се запознахме със семантиката на символните низове, как можем да ги създаваме и извеждаме, следва да се научим как да боравим с тях и да ги обработваме. Езикът C# ни дава набор от готови функции, които ще използваме за манипулация на стринговете.

Сравняване на низове по азбучен ред

Има множество начини за сравнение на символни низове и в зависимост от това какво точно ни е необходимо в конкретния случай, може да се възползваме от различните възможности на класа **string**.

Сравнение за еднаквост

Ако условието изисква да сравним два символни низа и да установим дали стойностите им са еднакви или не, удобен метод е методът **`Equals(…)`**, който работи еквивалентно на оператора **`==`**. Той връща булев резултат със стойност **`true`**, ако низовете имат еднакви стойности, и **`false`**, ако те са различни. Методът **`Equals(…)`** проверява за побуквено равенство на стойностите на низовете, като прави разлика между малки и главни букви. Т.е. сравняването на "c#" и "C#" с метода **`Equals(…)`** ще върне стойност **`false`**. Нека разгледаме един пример:

```
string word1 = "C#";
string word2 = "c#";
Console.WriteLine(word1.Equals("C#"));
Console.WriteLine(word1.Equals(word2));
Console.WriteLine(word1 == "C#");
Console.WriteLine(word1 == word2);

// Console output:
// True
// False
// True
// False
```

В практиката често ще ни интересува самото текстово съдържание при сравнение на два низа, без значение от регистъра (**casing**) на буквите. За да игнорираме разликата между малки и главни букви при сравнението на низове можем да използваме **`Equals(…)`** с параметър **`StringComparison.CurrentCultureIgnoreCase`**. Така в долния пример при сравнение на "C#" със "c#" методът ще върне стойност **`true`**:

```
Console.WriteLine(word1.Equals(word2,
  StringComparison.CurrentCultureIgnoreCase));
// True
```

`StringComparison.CurrentCultureIgnoreCase` е константа от изброения тип **`StringComparison`**. Какво е изброен тип и как можем да го използваме, ще научим в темата "Дефиниране на класове".

Сравнение на низове по азбучен ред

Вече стана ясно как сравняваме низове за еднаквост, но как ще установим лексикографската подредба на няколко низа? Ако пробваме да използваме операторите **<** и **>**, които работят отлично за сравнение на числа, ще установим, че те не могат да се използват за стрингове.

Ако искаме да сравним две думи и да получим информация коя от тях е преди другата, според азбучния ред на буквите в нея, на помощ идва методът **`CompareTo(…)`**. Той ни дава възможност да сравняваме стойностите на два символни низа и да установяваме лексикографската им наредба. За да бъдат два низа с еднакви стойности, те трябва да имат една и съща дължина (брой символи) и изграждащите ги символи трябва съответно да си съвпадат. Например низовете "give" и "given" са различни, защото имат различна дължина, а "near" и "fear" се различават по първия си символ.

Методът **`CompareTo(…)`** от класа **`String`** връща отрицателна стойност, 0 или положителна стойност в зависимост от лексикографската подредба на двата низа, които се сравняват. Отрицателна стойност означава, че първият низ е лексикографски преди втория, нула означава, че двата низа са еднакви, а положителна стойност означава, че вторият низ е лексикографски преди първия. За да си изясним по-добре как се сравняват лексикографски низове, нека разгледаме няколко примера:

```
string score = "sCore";
string scary = "scary";
Console.WriteLine(score.CompareTo(scary));
Console.WriteLine(scary.CompareTo(score));
Console.WriteLine(scary.CompareTo(scary));

// Console output:
// 1
// -1
// 0
```

Първият експеримент е извикването на метода **`CompareTo(…)`** на низа **`score`**, като подаден параметър е променливата **`scary`**. Първият символ връща знак за равенство. Тъй като методът **не игнорира** регистъра за малки и главни букви, още във втория символ открива несъответствие (в първия низ е "C", а във втория "c"). Това е достатъчно за определяне на подредбата на низовете и **`CompareTo(…)`** връща +1. Извикването на същия метод с разменени места на низовете връща -1, защото тогава отправната точка е низът **`scary`**. Последното му извикване логично връща 0, защото сравняваме **`scary`** със себе си.

Ако трябва да сравняваме низове лексикографски, игнорирайки регистъра на буквите, можем да използваме **`string.Compare(string strA, string strB, bool ignoreCase)`**. Това е статичен метод, който действа по същия начин както **`CompareTo(…)`**, но има опция **`ignoreCase`** за игнориране на регистъра за главни и малки букви. Нека разгледаме метода в действие:

```
string alpha = "alpha";
string score1 = "sCorE";
string score2 = "score";

Console.WriteLine(string.Compare(alpha, score1, false));
Console.WriteLine(string.Compare(score1, score2, false));
Console.WriteLine(string.Compare(score1, score2, true));
Console.WriteLine(string.Compare(score1, score2,
   StringComparison.CurrentCultureIgnoreCase));
// Console output:
// -1
// 1
// 0
// 0
```

В последния пример методът **`Compare(…)`** приема като трети параметър **`StringComparison.CurrentCultureIgnoreCase`** – вече познатата от метода **`Equals(…)`** константа, чрез която също можем да сравняваме низове, без да отчитаме разликата между главни и малки букви.

Забележете, че според методите **`Compare(…)`** и **`CompareTo(…)`** малките латински букви са лексикографски преди главните. Коректността на това правило е доста спорна, тъй като в Unicode таблицата главните букви са преди малките. Например според стандарта Unicode буквата "A" има код 65, който е по-малък от кода на буквата "a" (97).

Следователно, трябва да имате предвид, че лексикографското сравнение не следва подредбата на буквите в Unicode таблицата и при него могат да се наблюдават и други аномалии породени от особености на текущата култура.

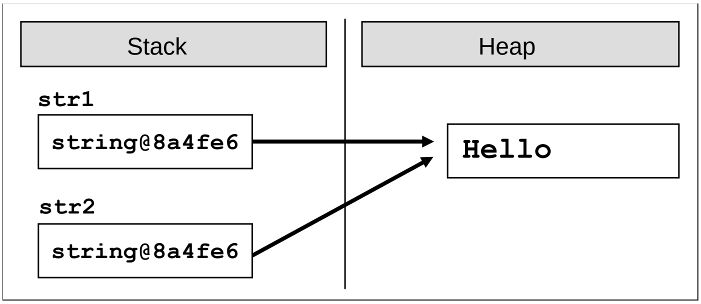
> ⚠ **Когато искате просто да установите дали стойностите на два символни низа са еднакви или не, използвайте метода `Equals(…)` или оператора `==`. Методите `CompareTo(…)` и `string.Compare(…)` са проектирани за употреба при лексикографска подредба на низове и не трябва да се използват за проверка за еднаквост.**

Операторите == и !=

В езика C# операторите == и != за символни низове работят чрез вътрешно извикване на **`Equals(…)`**. Ще прегледаме примери за използването на тези два оператора с променливи от тип символни низове:

```
string str1 = "Hello";
string str2 = str1;
Console.WriteLine(str1 == str2);
// Console output:
// True
```

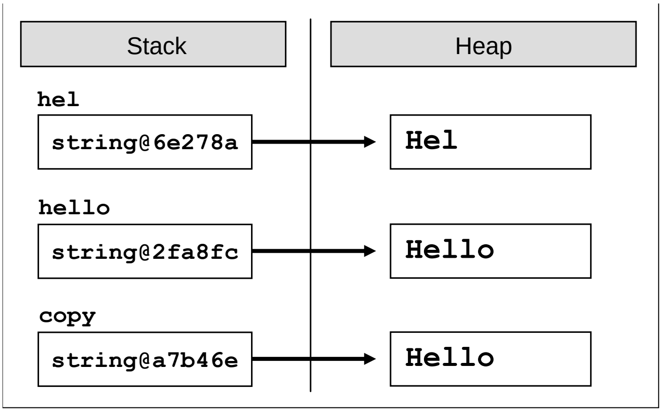
Сравнението на съвпадащите низове **`str1`** и **`str2`** връща стойност **`true`**. Това е напълно очакван резултат, тъй като насочваме променливата **`str2`** към мястото в динамичната памет, което е запазено за променливата **`str1`**. Така двете променливи имат един и същ адрес и проверката за равенство връща истина. Ето как изглежда паметта с двете променливи:


Stack
Heap
str1
string@8a4fe6
str2
string@8a4fe6
Hello

Да разгледаме още един пример:

```
string hel = "Hel";
string hello = "Hello";
string copy = hel + "lo";
Console.WriteLine(copy == hello);
// True
```

Обърнете внимание, че сравнението е между низовете **`hello`** и **`copy`**. Първата променлива директно приема стойността "**`Hello`**". Втората получава стойността си като резултат от съединяването на променлива и литерал, като крайният резултат е еквивалентен на стойността на първата променлива. В този момент двете променливи сочат към различни области от паметта, но съдържанието на съответните блокове памет е еднакво. Сравнението, направено с оператора **`==`** връща резултат **`true`**, въпреки че двете променливи сочат към различни области от паметта. Ето как изглежда паметта в този момент:


Stack
Heap
hel
string@6e278a
Hel
hello
string@2fa8fc
Hello
copy
string@a7b46e
Hello

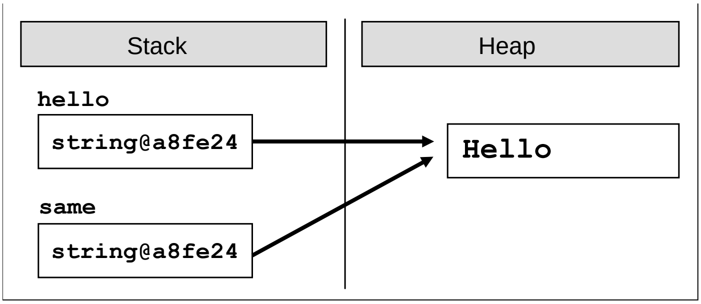
Оптимизация на паметта при символни низове

Нека видим следния пример:

```
string hello = "Hello";
string same = "Hello";
```

Създаваме първата променлива със стойност "**Hello**". Създаваме и втората променлива със стойност същия литерал. Логично е при създаването на променливата **hello** да се задели място в динамичната памет, да се запише стойността и променливата да сочи към въпросното място. При създаването на **same** също би трябвало да се създаде нова област, да се запише стойността и да се насочи препратката.

Истината обаче е, че съществува оптимизация в C# компилатора и в CLR, която спестява създаването на дублирани символни низове в паметта. Тази оптимизация се нарича **интерниране на низовете** (**strings interning**) и благодарение на нея двете променливи в паметта ще сочат към един и същ общ блок от паметта. Това намалява разхода на памет и оптимизира някои операции, например сравнението на такива напълно съвпадащи низове. Те се записват в паметта по следния начин:

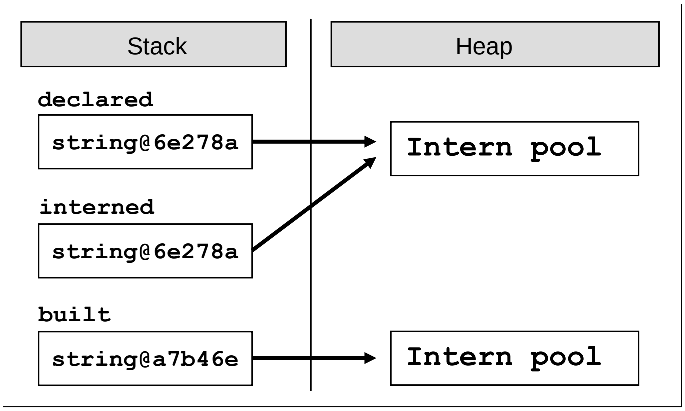


Когато инициализираме променлива от тип **string** с низов литерал, скрито от нас динамичната памет се обхожда и се прави проверка дали такава стойност вече съществува. Ако съществува, новата променлива просто се пренасочва към нея. Ако не, заделя се нов блок памет, стойността се записва в него и референцията се препраща да сочи към новия блок. Интернирането на низове в .NET е възможно, защото низовете по концепция са неизменими и няма опасност някой да промени областта, сочена от няколко стрингови променливи едновременно.

Когато не инициализираме низовете с литерали, не се ползва интерниране. Все пак, ако искаме да използваме интерниране изрично, можем да го направим чрез метода **Intern(…)**:

```
string declared = "Intern pool";
string built = new StringBuilder("Intern pool").ToString();
string interned = string.Intern(built);
Console.WriteLine(object.ReferenceEquals(declared, built));
Console.WriteLine(object.ReferenceEquals(declared, interned));
// Output:
// False
// True
```

Ето и състоянието на паметта в този момент:



В примера ползвахме статичния метод **`Object.ReferenceEquals(…)`**, който сравнява два обекта в паметта и връща дали сочат към един и същи блок памет. Ползвахме и класа **`StringBuilder`**, който служи за ефективно построяване на низове. Кога и как се ползва **`StringBuilder`** ще обясним в детайли след малко, а сега нека се запознаем с основните операции върху низове.

Операции за манипулация на символни низове

След като се запознахме с основите на символните низове и тяхната структура, идва ред на средствата за тяхната обработка. Ще прегледаме слепването на текстови низове, търсене в съдържанието им, извличане на поднизове и други операции, които ще ни послужат при решаване на различни проблеми от практиката.

⚠	**Символните низове са неизменими! Всяка промяна на променлива от тип `string` създава нов низ, в който се записва резултатът. По тази причина операциите, които прилагате върху символните низове, връщат като резултат референция към получения резултат.**

Възможна е и обработката на символни низове без създаването на нови обекти в паметта при всяка модификация, но за целта трябва да се използва класът **`StringBuilder`**, с който ще се запознаем след малко.

Долепване на низове (конкатенация)

Долепването на символни низове и получаването на нов низ като резултат, се нарича **конкатенация**. То може да бъде извършено по няколко начина: чрез метода **`Concat(…)`** или чрез операторите **+** и **+=**.

Пример за използване на функцията **`Concat(…)`**:

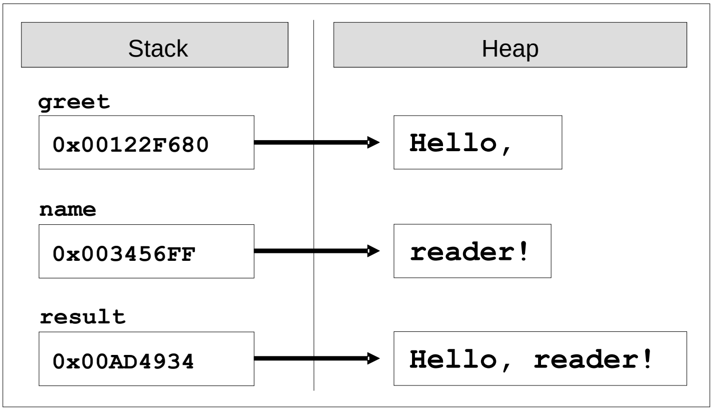
```
string greet = "Hello, ";
string name = "reader!";
string result = string.Concat(greet, name);
```

Извиквайки метода, ще долепим променливата **`name`**, която е подадена като аргумент, към променливата **`greet`**. Резултатният низ ще има стойност "**`Hello, reader!`**".

Вторият вариант за конкатенация е чрез операторите **+** и **+=**. Горният пример може да реализираме още по следния начин:

```
string greet = "Hello, ";
string name = "reader!";
string result = greet + name;
```

И в двата случая в паметта тези променливи ще се представят по следния начин:



Обърнете внимание, че долепванията на низове не променят стойностите на съществуващите променливи, а връщат нова променлива като резултат. Ако опитаме да долепим два стринга, без да ги запазим в променлива, промените нямат да бъдат съхранени. Ето една типична грешка:

```
string greet = "Hello, ";
string name = "reader!";
string.Concat(greet, name);
```

В горния пример двете променливи се слепват, но резултатът от слепването не се съхранява никъде и се губи.

Ако искаме да добавим някаква стойност към съществуваща променлива, например променливата **`result`**, с познатите ни оператори можем да ползваме следния код:

```
result = result + " How are you?";
```

За да си спестим повторното писане на декларираната по-горе променлива, можем да използваме оператора **+=**:

```
result += " How are you?";
```

И в двата случая резултатът ще бъде един и същ: "**Hello, reader! How are you?**".

Към символните низове можем да долепим и други данни, които могат да бъдат представени в текстов вид. Възможна е конкатенацията с числа, символи, дати и др. Ето един пример:

```
string message = "The number of the beast is: ";
int beastNum = 666;
string result = message + beastNum;
// The number of the beast is: 666
```

Както виждаме от горния пример, няма проблем да съединяваме символни низове с други данни, които не са от тип **string**. Нека прегледаме още един, пълен пример за слепването на символни низове:

```
public class DisplayUserInfo
{
  static void Main()
  {
    string firstName = "Svetlin";
    string lastName = "Nakov";
    string fullName = firstName + " " + lastName;

    int age = 28;
    string nameAndAge = "Name: " + fullName + "\nAge: " + age;
    Console.WriteLine(nameAndAge);
  }
}
// Console output:
// Name: Svetlin Nakov
// Age: 28
```

Преминаване към главни и малки букви

Понякога имаме нужда да променим съдържанието на символен низ, така че всички символи в него да бъдат само с главни или малки букви. Двата метода, които биха ни свършили работа в случая, са **ToLower(…)** и **ToUpper(…)**. Първият от тях конвертира всички главни букви към малки:

```
string text = "All Kind OF LeTTeRs";
Console.WriteLine(text.ToLower());
// all kind of letters
```

От примера се вижда, че всички главни букви от текста сменят регистъра си и целият текст преминава изцяло в малки букви. Такова преминаване към малки букви е удобно например при съхраняване на потребителските имена (username) в различни онлайн системи. При регистрация потребителите могат да ползват смесица от главни и малки букви, но системата след това може да ги направи всичките малки за да ги унифицира и да избегне съвпадения по букви с разлики в регистъра.

Ето още един пример. Искаме да сравним въведен от потребителя вход и не сме сигурни по какъв точно начин е написан той – с малки или главни букви или смесено. Един възможен подход е да уеднаквим регистъра на буквите и да го сравним с дефинираната от нас константа. По този начин не правим разлика за малки и главни букви. Например ако имаме входен панел на потребителя, в който въвеждаме име и парола, и няма значение дали паролата е написана с малки или главни букви, може да направим подобна проверка на паролата:

```
string pass1 = "Parola";
string pass2 = "PaRoLa";
string pass3 = "parola";
Console.WriteLine(pass1.ToUpper() == "PAROLA");
Console.WriteLine(pass2.ToUpper() == "PAROLA");
Console.WriteLine(pass3.ToUpper() == "PAROLA");

// Console output:
// True
// True
// True
```

В примера сравняваме три пароли с еднакво съдържание, но с различен регистър. При проверката съдържанието им се проверява дали съвпада побуквено със символния низ "PAROLA". Разбира се, горната проверка бихме могли да направим и чрез метода **`Equals(…)`** във варианта с игнориране на регистъра на символите, който вече разгледахме.

Търсене на низ в друг низ

Когато имаме символен низ със зададено съдържание, често се налага да обработим само част от стойността му. .NET платформата ни предоставя два метода за търсене на низ в друг низ: **`IndexOf(…)`** и **`LastIndexOf(…)`**. Те претърсват даден символен низ и проверяват дали подаденият като параметър подниз се среща в съдържанието му. Връщаният резултат от методите е цяло число. Ако резултатът е неотрицателна стойност, тогава това е позицията, на която е открит първият символ от подниза. Ако методът върне стойност -1, това означава, че поднизът не е открит. Напомняме, че в C# индексите на символите в низовете започват от 0.

Методите **`IndexOf(…)`** и **`LastIndexOf(…)`** претърсват съдържанието на текстова последователност, но в различна посока. Търсенето при първия метод започва от началото на низа в посока към неговия край, а при втория метод – търсенето се извършва отзад напред. Когато се интересуваме от първото срещнато съвпадение, използваме **`IndexOf(…)`**. Ако искаме да претърсваме низа от неговия край (например за откриване на последната точка в името на даден файл или последната наклонена черта в URL адрес), ползваме **`LastIndexOf(…)`**.

При извикването на **`IndexOf(…)`** и **`LastIndexOf(…)`** може да се подаде и втори параметър, който указва от коя позиция да започне търсенето. Това е полезно, ако искаме да претърсваме част от даден низ, а не целия низ.

Търсене в символен низ – пример

Да разгледаме един пример за използване на метода **`IndexOf(…)`**:

```
string book = "Introduction to C# book";
int index = book.IndexOf("C#");
Console.WriteLine(index);
// index = 16
```

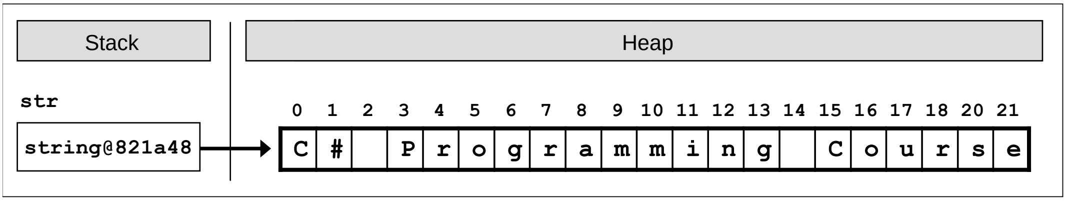
В примера променливата **`book`** има стойност "Introduction to C# book". Търсенето на подниза "C#" в тази променлива ще върне стойност 16, защото поднизът ще бъде открит в стойността на отправната променлива и първият символ "C" от търсената дума се намира на 16-та позиция.

Търсене с IndexOf(…) – пример

Нека прегледаме още един, по-подробен пример за търсенето на отделни символи и символни низове в текст:

```
string str = "C# Programming Course";

int index = str.IndexOf("C#"); // index = 0
index = str.IndexOf("Course"); // index = 15
index = str.IndexOf("COURSE"); // index = -1
index = str.IndexOf("ram");    // index = 7
index = str.IndexOf("r");      // index = 4
index = str.IndexOf("r", 5);   // index = 7
index = str.IndexOf("r", 10);  // index = 18
```

Ето как изглежда в паметта символният низ, в който търсим:



Ако обърнем внимание на резултата от третото търсене, ще забележим, че търсенето на думата "**`COURSE`**" в текста връща резултат -1, т.е. няма намерено съответствие. Въпреки че думата се намира в текста, тя е написана с различен регистър на буквите. Методите **`IndexOf(…)`** и **`LastIndexOf(…)`** правят разлика между малки и главни букви. Ако искаме да игнорираме тази разлика, можем да запишем текста в нова променлива и да го превърнем към текст с изцяло малки или изцяло главни букви, след което да извършим търсене в него, независещо от регистъра на буквите.

Всички срещания на дадена дума – пример

Понякога искаме да открием всички срещания на даден подниз в друг низ. Използването на двата метода само с един подаден аргумент за търсен низ не би ни свършило работа, защото винаги ще връща само първото срещане на подниза. Можем да подаваме втори параметър за индекс, който посочва началната позиция, от която да започва търсенето. Разбира се, ще трябва да завъртим и цикъл, с който да се придвижваме от първото срещане на търсения низ към следващото, по-следващото и т.н. до последното.

Ето един пример за използването на **`IndexOf(…)`** по дадена дума и начален индекс: откриване на всички срещания на думата "C#" в даден текст:

```
string quote = "The main intent of the \"Intro C#\"" +
  " book is to introduce the C# programming to newbies.";
string keyword = "C#";
int index = quote.IndexOf(keyword);

while (index != -1)
{
  Console.WriteLine("{0} found at index: {1}", keyword, index);
  index = quote.IndexOf(keyword, index + 1);
}
```

Първата стъпка е да направим търсене за ключовата дума "C#". Ако думата е открита в текста (т.е. връщаната стойност е различна от -1), извеждаме я на конзолата и продължаваме търсенето надясно, започвайки от позицията, на която сме открили думата, увеличена с единица. Повтаряме действието, докато **`IndexOf(…)`** върне стойност -1.

Забележка: ако на последния ред пропуснем задаването на начален индекс, то търсенето винаги ще започва отначало и ще връща една и съща стойност. Това ще доведе до зацикляне на програмата ни. Ако пък търсим директно от индекса, без да увеличаваме с единица, ще попадаме всеки път отново и отново на последния резултат, чийто индекс сме вече намерили. Ето защо правилното търсене на следващ резултат трябва да започва от начална позиция **`index + 1`**.

Извличане на част от низ

За момента знаем как да проверим дали даден подниз се среща в даден текст и на кои позиции се среща. Как обаче да извлечем част от низа в отделна променлива?

Решението на проблема ни е методът **`Substring(…)`**. Използвайки го, можем да извлечем дадена част от низ (подниз) по зададени начална позиция в текста и дължина. Ако дължината бъде пропусната, ще бъде направена извадка от текста, започваща от началната позиция до неговия край.

Следва пример за извличане на подниз от даден низ:

```
string path = "C:\\Pics\\Rila2010.jpg";
string fileName = path.Substring(8, 8);
// fileName = "Rila2010"
```

Променливата, която манипулираме, е **`path`**. Тя съдържа пътя до файл от файловата ни система. За да присвоим името на файла на нова променлива, използваме **`Substring(8, 8)`** и взимаме последователност от 8 символи, стартираща от позиция 8, т.е. символите на позиции от 8 до 15.

Извикването на метода `Substring(startIndex, length)` извлича подниз от даден стринг, който се намира между `startIndex` и `(startIndex + length – 1)` включително. Символът на позицията `startIndex + length` не се взима предвид! Например ако посочим `Substring(8, 3)`, ще бъдат извлечени символите между индекс 8 и 10 включително.

Ето как изглеждат символите, които съставят текста, от който извличаме подниз:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
C	**:**	\	**P**	**i**	**c**	**s**	\	**R**	**i**	**l**	**a**	**2**	**0**	**1**	**0**	**.**	**j**	**p**	**g**

Придържайки се към схемата, извикваният метод трябва да запише символите от позиции от 8 до 15 включително (тъй като последният индекс не се включва), а именно "**`Rila2010`**".

Да разгледаме една по-интересна задача. Как бихме могли да изведем името на файла и неговото разширение? Тъй като знаем как се записва път във файловата система, можем да процедираме по следния план:

- Търсим последната обратна наклонена черта в текста;
- Записваме позицията на последната наклонена черта;
- Извличаме подниза, започващ от **получената позиция + 1**.

Да вземем отново за пример познатия ни **`path`**. Ако нямаме информация за съдържанието на променливата, но знаем, че тя съдържа път до файл, може да се придържаме към горната схема:

```
string path = "C:\\Pics\\Rila2010.jpg";
int index = path.LastIndexOf("\\");
// index = 7
string fullName = path.Substring(index + 1);
// fullName = "Rila2010.jpg"
```

Разцепване на низ по разделител

Един от най-гъвкавите методи за работа със символни низове е **`Split(…)`**. Той ни дава възможност да разцепваме един низ по разделител или масив от възможни разделители. Например можем да обработваме променлива, която има следното съдържание:

```
string listOfBeers = "Amstel, Zagorka, Tuborg, Becks";
```

Как можем да отделим всяка една бира в отделна променлива или да запишем всички бири в масив? На пръв поглед може да изглежда трудно – трябва да търсим с **`IndexOf(…)`** за специален символ, след това да отделяме подниз със **`Substring(…)`**, да итерираме всичко това в цикъл и да записваме резултата в дадена променлива. Тъй като разделянето на низ по разделител е основна задача от текстообработката, в .NET Framework има готови методи за това.

Разделяне на низ по множество от разделители – пример

По-лесния и гъвкав начин да разрешим проблема е следният:

```
char[] separators = new char[] {' ', ',', '.'};
string[] beersArr = listOfBeers.Split(separators);
```

Използвайки вградената функционалност на метода **`Split(…)`** от класа **`String`**, ще разделим съдържанието на даден низ по масив от символи-разделители, които са подадени като аргумент на метода. Всички поднизове, между които присъстват интервал, запетая или точка, ще бъдат отделени и записани в масива **`beersArr`**.

Ако обходим масива и изведем елементите му един по един, резултатите ще бъдат: "Amstel", "", "Zagorka", "", "Tuborg", "" и "Becks". Получаваме 7 резултата, вместо очакваните 4. Причината е, че при разделянето на текста се откриват 3 подниза, които съдържат два разделителни символа един до друг (например запетая, последвана от интервал). В този случай празният низ между двата разделителя също е част от връщания резултат.

Как да премахнем празните елементи?

Ако искаме да игнорираме празните низове, едно възможно разрешение е да правим проверка при извеждането им:

```
foreach (string beer in beersArr)
{
   if (beer != "")
   {
      Console.WriteLine(beer);
   }
}
```

С този подход обаче не премахваме празните низове от масива, а просто не ги отпечатваме. Затова можем да променим аргументите, които подаваме на метода **`Split(…)`**, като подадем една специална опция:

```
string[] beersArr = listOfBeers.Split(
  separators, StringSplitOptions.RemoveEmptyEntries);
```

След тази промяна масивът **`beersArr`** ще съдържа 4 елемента – четирите думи от променливата **`listOfBeers`**.

При разделяне на низове добавяйки като втори параметър константата `StringSplitOptions.RemoveEmptyEntries` ние инструктираме метода `Split(…)` да работи по следния начин: "Върни всички поднизове от променливата, които са разделени от интервал, запетая или точка. Ако срещнеш два или повече съседни разделителя, считай ги за един".

Замяна на подниз с друг

Текстообработката в .NET Framework предлага готови методи за замяна на един подниз с друг. Например ако сме допуснали една и съща техническа грешка при въвеждане на email адреса на даден потребител в официален документ, можем да го заменим с помощта на метода **`Replace(…)`**:

```
string doc = "Hello, some@gmail.com, " +
  "you have been using some@gmail.com in your registration.";
string fixedDoc =
  doc.Replace("some@gmail.com", "osama@bin-laden.af");
Console.WriteLine(fixedDoc);

// Console output:
// Hello, osama@bin-laden.af, you have been using
// osama@bin-laden.af in your registration.
```

Както се вижда от примера, методът **`Replace(…)`** замества всички срещания на даден подниз с даден друг подниз, а не само първото.

Регулярни изрази

Регулярните изрази (regular expressions) са мощен инструмент за обработка на текст и позволяват търсене на съвпадения по **шаблон** (**pattern**). Пример за шаблон е **[A-Z0-9]+**, който означава непразна поредица от главни латински букви и цифри. Регулярните изрази позволяват по-лесна и по-прецизна обработка на текстови данни: извличане на определени ресурси от текстове, търсене на телефонни номера, откриване на електронна поща в текст, разделяне на всички думи в едно изречение, валидация на данни и т.н.

Регулярни изрази – пример

Ако имаме служебен документ, който се използва само в офиса, и в него има лични данни, трябва да ги цензурираме, преди да ги пратим на клиента. Например можем да цензурираме всички номера на мобилни телефони и да ги заместим със звездички. Използвайки регулярните изрази, това би могло да стане по следния начин:

```
string doc = "Smith's number: 0898880022\nFranky can be " +
  " found at 0888445566.\nSteven’ mobile number: 0887654321";
string replacedDoc = Regex.Replace(
  doc, "(08)[0-9]{8}", "$1********");
Console.WriteLine(replacedDoc);

// Console output:
// Smith's number: 08********
// Franky can be  found at 08********.
// Steven' mobile number: 08********
```

Обяснение на аргументите на Regex.Replace(…)

В горния фрагмент от код използваме регулярен израз, с който откриваме всички телефонни номера в зададения ни текст и ги заменяме по шаблон. Използваме класа **System.Text.RegularExpressions.Regex**, който е предназначен за работа с регулярни изрази в .NET Framework. Променливата, която имитира документа с текстовите данни, е **doc**. В нея са записани няколко имена на клиенти заедно с техните телефонни номера. Ако искаме да предпазим контактите от неправомерно използване и желаем да цензурираме телефонните номера, то може да заменим всички мобилни телефони със звездички. Приемайки, че телефоните са записани във формат: "**08 + 8 цифри**", методът **Regex.Replace(…)** открива всички съвпадения по дадения формат и ги замества с: "08********".

Регулярният израз, отговорен за откриването на номерата, е следният: "**(08)[0-9]{8}**". Той намира всички поднизове в текста, изградени от константата "**08**" и следвани от точно 8 символа в диапазона от 0 до 9. Примерът може да бъде допълнително подобрен за подбиране на номерата само от дадени мобилни оператори, за работа с телефони на чуждестранни мрежи и др., но в случая използван опростен вариант.

Литералът "**08**" е заграден от кръгли скоби. Те служат за обособяване на отделна група в регулярния израз. Групите могат да бъдат използвани за обработка само на определена част от израза, вместо целия израз. В нашия пример, групата е използвана в заместването. Чрез нея откритите съвпадения се заместват по шаблон "$1********", т.е. текстът намерен от първата група на регулярния израз (**$1**) + последователни 8 звездички за цензурата. Тъй като дефинираната от нас група винаги е константа (08), то заместеният текст ще бъде винаги: 08********.

Настоящата тема няма за цел да обясни как се работи с регулярни изрази в .NET Framework, тъй като това е голяма и сложна материя, а само да обърне внимание на читателя, че регулярните изрази съществуват и са много мощно средство за текстообработка. Който се интересува повече, може да потърси статии, книги и самоучители, от които да разучи как се конструират регулярните изрази, как се търсят съвпадения, как се прави валидация, как се правят замествания по шаблон и т.н. По-конкретно препоръчваме да посетите сайтовете http://www.regular-expressions.info/ и http://regexlib.com/. Повече информация за класовете, които .NET Framework предлага за работа с регулярни изрази и как точно се използват, може да бъде открита на адрес: http://msdn.microsoft.com/en-us/library/system.text.regularexpressions.regex%28VS.100%29.aspx.

Премахване на ненужни символи в началото и в края на низ

Въвеждайки текст във файл или през конзолата, понякога се появяват "паразитни" **празни места** (**white-space**) в началото или в края на текста – някой друг интервал или табулация, които да може да не се доловят на пръв поглед. Това може да не е съществено, но ако не валидираме потребителски данни, би било проблем от гледна точка на проверка съдържанието на входната информация. За решаване на проблема на помощ идва методът **`Trim()`**. Той се грижи именно за премахването на паразитните празни места в началото или края на даден символен низ. Празните места могат да бъдат интервали, табулация, нови редове и др.

Нека в променливата **`fileData`** сме прочели съдържанието на файл, в който е записано име на студент. Пишейки текста или преобръщайки го от един формат в друг може да са се появили паразитни празни места и тогава променливата ще изглежда по подобен начин:

```
string fileData = "   \n\n      Ivan Ivanov      ";
```

Ако изведем съдържанието на конзолата, ще получим 2 празни реда, последвани от няколко интервала, търсеното от нас име и още няколко допълнителни интервала в края. Можем да редуцираме информацията от променливата само до нужното ни име по следния начин:

```
string reduced = fileData.Trim();
```

Когато изведем повторно информацията на конзолата, съдържанието ще бъде "**`Ivan Ivanov`**", без нежеланите празни места.

Премахване на ненужни символи по зададен списък

Методът **`Trim(…)`** може да приема масив от символи, които искаме да премахнем от низа. Това може да направим по следния начин:

```
string fileData = "   111 $  %    Ivan Ivanov  ### s   ";
char[] trimChars = new char[] {' ', '1', '$', '%', '#', 's'};
string reduced = fileData.Trim(trimChars);
// reduced = "Ivan Ivanov"
```

Отново получаваме желания резултат "**`Ivan Ivanov`**".

Обърнете внимание, че трябва да изброим всички символи, които искаме да премахнем, включително празните интервали (интервал, табулация, нов ред и др.). Без наличието на `' '` в масива `trimChars`, нямаше да получим желания резултат!

Ако искаме да премахнем паразитните празни места само в началото или в края на низа, можем да използваме методите **`TrimStart(…)`** и **`TrimEnd(…)`**:

```
string reduced = fileData.TrimEnd(trimChars);
// reduced = "   111 $  %    Ivan Ivanov"
```

Построяване на символни низове. StringBuilder

Както обяснихме по-горе, символните низове в C# са неизменими. Това означава, че всички корекции, приложени върху съществуващ низ, не го променят, а връщат като резултат нов символен низ. Например използването на методите **`Replace(…)`**, **`ToUpper(…)`**, **`Trim(…)`** не променят низа, за който са извикани, а **заделят нова област от паметта**, в която се записва новото съдържание. Това има много предимства, но в някои случаи може да ни създаде проблеми с производителността.

Долепяне на низове в цикъл: никога не го правете!

Сериозен проблем с производителността може да срещнем, когато се опитаме да конкатенираме символни низове в цикъл. Проблемът е пряко свързан с обработката на низовете и динамичната памет, в която се съхраняват те. За да разберем как се получава недостатъчното бързодействие при съединяване на низове в цикъл, трябва първо да разгледаме какво се случва при използване на оператора "+" за низове.

Как работи съединяването на низове?

Вече се запознахме с начините за съединяване на низове в C#. Нека сега разгледаме какво се случва в паметта, когато се съединяват низове. Да вземем за пример две променливи **`str1`** и **`str2`** от тип **`string`**, които имат стойности съответно "Super" и "Star". В хийпа (динамичната памет) има заделени две области, в които се съхраняват стойностите. Задачата на **`str1`** и **`str2`** е да пазят препратка към адресите в паметта, на които се намират записаните от нас данни. Нека създадем променлива **`result`** и ѝ придадем стойността на другите два низа чрез долепяне. Фрагментът от код за създаването и дефинирането на трите променливи би изглеждал по следния начин:

```
string str1 = "Super";
string str2 = "Star";
string result = str1 + str2;
```

Какво ще се случи с паметта? Създаването на променливата `result` ще задели нова област от динамичната памет, в която ще запише резултата от `str1 + str2`, който е "SuperStar". След това самата променлива ще пази адреса на заделената област. Като резултат ще имаме три области в паметта, както и три референции към тях. Това е удобно, но създаването на нова област, записването на стойност, създаването на нова променлива и реферирането ѝ към паметта е времеотнемащ процес, който би бил проблем при многократното му повтаряне в цикъл.

За разлика от други езици за програмиране, в C# не е необходимо ръчното освобождаване на обектите, записани в паметта. Съществува специален механизъм, наречен **garbage collector (система за почистване на паметта)**, който се грижи за изчистването на неизползваната памет и ресурси. Системата за почистване на паметта е отговорна за освобождаването на обектите в динамичната памет, когато вече не се използват. Създаването на много обекти, придружени с множество референции в паметта, е вредно, защото така се запълва паметта и тогава автоматично се налага изпълнение на garbage collector. Това отнема немалко време и забавя цялостното изпълнение на процеса. Освен това преместването на символи от едно място на паметта в друго, което се изпълнява при съединяване на низове, е бавно, особено ако низовете са дълги.

Защо долепянето на низове в цикъл е лоша практика?

Да приемем, че имаме за задача да запишем числата от 1 до 20 000 последователно едно до друго в променлива от тип `string`. Как можем да решим задачата с досегашните си знания? Един от най-лесните начини за имплементация е създаването на променливата, която съхранява числата, и завъртането на цикъл от 1 до 20 000, в който всяко число се долепва към въпросната променлива. Реализирано на C#, решението би изглеждало примерно така:

```
string collector = "Numbers: ";
for (int index = 1; index <= 20000; index++)
{
  collector += index;
}
```

Изпълнението на горния код ще завърти цикъла 20 000 пъти, като след всяко завъртане ще добавя текущия индекс към променливата `collector`. Стойността на променливата `collector` след края на изпълнението ще бъде: "Numbers: 12345678910111213141516..." (останалите числа от 17 до 20 000 са заместени с многоточие, с цел относителна представа за резултата).

Вероятно не ви е направило впечатление забавянето при изпълнение на фрагмента. Всъщност използването на конкатенацията в цикъл е забавила значително нормалния изчислителен процес и на средностатистически компютър (от януари 2010 г.) итерацията на цикъла отнема 1-3 секунди. Потребителят на програмата ни би бил доста скептично настроен, ако се налага да чака няколко секунди за нещо елементарно, като слепване на числата от 1 до 20 000. Освен това в случая 20 000 е само примерна крайна точка. Какво ли ще бъде забавянето, ако вместо 20 000, потребителят има нужда да долепи числата до 200 000? Пробвайте!

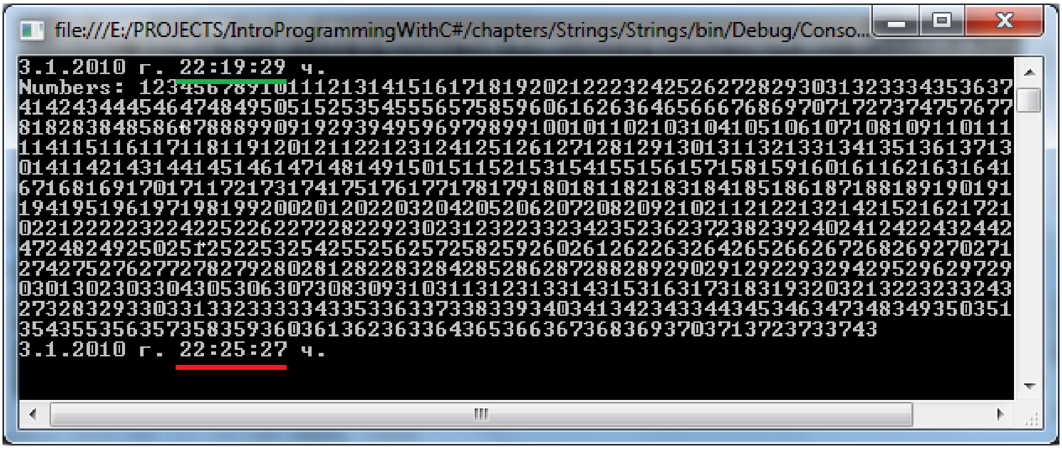
Конкатениране в цикъл с 200000 итерации - пример

Нека развием горния пример. Първо, ще променим крайната точка на цикъла от 20 000 на 200 000. Второ, за да отчетем правилно времето за изпълнение, ще извеждаме на конзолата текущата дата и час преди и след изпълнението на цикъла. Трето, за да видим, че променливата съдържа желаната от нас стойност, ще изведем част от нея на конзолата. Ако искате да се уверите, че цялата стойност е запаметена, може да премахнете прилагането на метода **`Substring(…)`**, но самото отпечатване в този случай също ще отнеме доста време.

Крайният вариант на примера би изглеждал така:

```
class NumbersConcatenator
{
   static void Main()
   {
      Console.WriteLine(DateTime.Now);

      string collector = "Numbers: ";
      for (int index = 1; index <= 200000; index++)
      {
         collector += index;
      }

      Console.WriteLine(collector.Substring(0, 1024));
      Console.WriteLine(DateTime.Now);
   }
}
```

При изпълнението на примера на конзолата се извеждат дата и час на стартиране на програмата, отрязък от първите 1024 символа от променливата, както и дата и час на завършване на програмата. Причината да покажем само първите 1024 символа е, че искаме да измерим само времето за изчисленията без времето за отпечатване на резултата. Нека видим примерния изход от изпълнението:


file:///E:/PROJECTS/IntroProgrammingWithC#/chapters/Strings/Strings/bin/Debug/Conso...

```
3.1.2010 г. 22:19:29 ч.
Numbers: 12345678910111213141516171819202122232425262728293031323334353637
41424344454647484950515253545556575859606162636465666768697071727374757677
81828384858687888990919293949596979899100101102103104105106107108109110111
11411511611711811912012112212312412512612712812913013113213313413513613713
01411421431441451461471481491501511521531541551561571581591601611621631641
67168169170171172173174175176177178179180181182183184185186187188189190191
19419519619719819920020120220320420520620720820921021121221321421521621721
02212222232242252262272282292302312322332342352362372382392402412422432442
47248249250251252253254255256257258259260261262263264265266267268269270271
27427527627727827928028128228328428528628728828929029129229329429529629729
03013023033043053063073083093103113123133143153163173183193203213223233243
27328329330331332333334335336337338339340341342343344345346347348349350351
3543553563573583593603613623633643653663673683693703713723733743
3.1.2010 г. 22:25:27 ч.
```

Със зелена линия е подчертан часът в началото на изпълнението на програмата, а с червена – нейният край. Обърнете внимание на времето за изпълнение – почти 6 минути! Подобно изчакване е недопустимо за подобна задача и не само ще изнерви потребителя, а ще го накара да спре програмата без да я изчака до край.

Обработка на символни низове в паметта

Проблемът с времеотнемащата обработка на цикъла е свързан именно с работата на низовете в паметта. Всяка една итерация създава нов обект в динамичната памет и насочва референцията към него. Процесът изисква определено физическо време.

На всяка стъпка се случват няколко неща:

1. Заделя се област от паметта за записване на резултата от долепването на поредното число. Тази памет се използва само временно, докато се изпълнява долепването, и се нарича **буфер**.
2. Премества се старият низ в новозаделения буфер. Ако низът е дълъг (примерно 1 MB или 10 MB), това може да е доста бавно!
3. Долепя се поредното число към буфера.
4. Буферът се преобразува в символен низ.
5. Старият низ, както и временният буфер, остават неизползвани и по някое време биват унищожени от системата за почистване на паметта (**garbage collector**). Това също може да е бавна операция.

Много по-елегантен и удачен начин за конкатениране на низове в цикъл е използването на класа `StringBuilder`. Нека видим как става това.

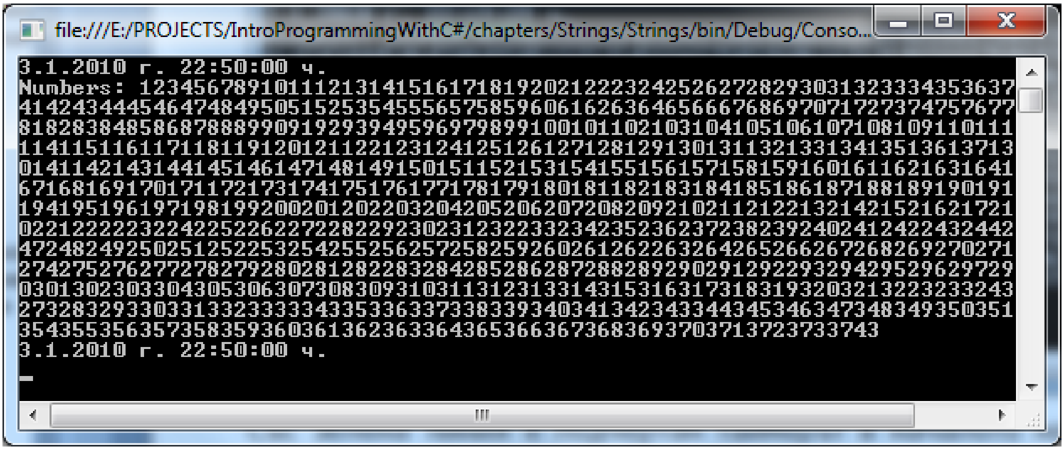
Построяване и промяна на низове със StringBuilder

`StringBuilder` е клас, който служи за построяване и промяна на символни низове. Той преодолява проблемите с бързодействието, които възникват при конкатениране на низове от тип **`string`**. Класът е изграден под формата на масив от символи и това, което трябва да знаем за него е, че информацията в него може свободно да се променя. Промените, които се налагат в променливите от тип **`StringBuilder`**, се извършват в една и съща област от паметта (буфер), което спестява време и ресурси. За промяната на съдържанието не се създава нов обект, а просто се променя текущият.

Нека пренапишем горния код, в който слепвахме низове в цикъл. Ако си спомняте, операцията отне 6 минути. Нека измерим колко време ще отнеме същата операция, ако използваме **`StringBuilder`**:

```
class ElegantNumbersConcatenator
{
   static void Main()
   {
      Console.WriteLine(DateTime.Now);

      StringBuilder sb = new StringBuilder();
      sb.Append("Numbers: ");

      for (int index = 1; index <= 200000; index++)
      {
         sb.Append(index);
      }

      Console.WriteLine(sb.ToString().Substring(0, 1024));
      Console.WriteLine(DateTime.Now);
   }
}
```

Примерът е базиран на предходния, със съвсем леки корекции. Връщаният резултат е същият, а какво ще кажете за времето за изпълнение?


file:///E:/PROJECTS/IntroProgrammingWithC#/chapters/Strings/Strings/bin/Debug/Conso...

```
3.1.2010 г. 22:50:00 ч.
Numbers: 1234567891011121314151617181920212223242526272829303132333435363 7
41424344454647484950515253545556575859606162636465666768697071727374757677
81828384858687888990919293949596979899100101102103104105106107108109110111
11411511611711811912012112212312412512612712812913013113213313413513613713
01411421431441451461471481491501511521531541551561571581591601611621631641
67168169170171172173174175176177178179180181182183184185186187188189190191
19419519619719819920020120220320420520620720820921021121221321421521621721
02212222232242252262272282292302312322332342352362372382392402412422432442
47248249250251252253254255256257258259260261262263264265266267268269270271
27427527627727827928028128228328428528628728828929029129229329429529629729
03013023033043053063073083093103113123133143153163173183193203213223233243
27328329330331332333334335336337338339340341342343344345346347348349350351
3543553563573583593603613623633643653663673683693703713723733743
3.1.2010 г. 22:50:00 ч.
_
```

Необходимото време за слепване на 200 000 символа със **`StringBuilder`** е вече по-малко от секунда!

Обръщане на низ на обратно – пример

Да разгледаме друг пример, в който искаме да обърнем съществуващ символен низ на обратно (отзад напред). Например ако имаме низа "abcd", върнатият резултат трябва да бъде "dcba". Взимаме първоначалния низ, обхождаме го отзад-напред символ по символ и добавяме всеки символ към променлива от тип **`StringBuilder`**:

```
public class WordReverser
{
  public static void Main()
  {
    string text = "EM edit";
    string reversed = ReverseText(text);
    Console.WriteLine(reversed);

    // Console output:
    // tide ME
  }

  public static string ReverseText(string text)
  {
    StringBuilder sb = new StringBuilder();
    for (int i = text.Length - 1; i >= 0; i--)
      sb.Append(text[i]);
    return sb.ToString();
  }
}
```

В демонстрацията имаме променливата **`text`**, която съдържа стойността "EM edit". Подаваме променливата на метода **`ReverseText(…)`** и приемаме новата стойност в променлива с име **`reversed`**. Методът, от своя страна, обхожда символите от променливата в обратен ред и ги записва в нова променлива от тип **`StringBuilder`**, но вече наредени обратно. В крайна сметка резултатът е "tide ME".

Как работи класът StringBuilder?

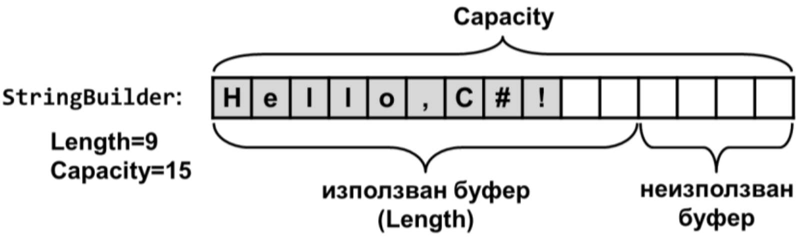
Класът **`StringBuilder`** представлява реализация на символен низ в C#, но различна от тази на класа **`string`**. За разлика от познатите ни вече символни низове, обектите на класа **`StringBuilder`** не са неизменими, т.е. редакциите не налагат създаването на нов обект в паметта. Това намалява излишното прехвърляне на данни в паметта при извършване на основни операции, като например долепяне на низ в края.

`StringBuilder` поддържа буфер с определен капацитет (по подразбиране 16 символа). Буферът е реализиран под формата на масив от символи, който е предоставен на програмиста с удобен интерфейс – методи за лесно и бързо добавяне и редактиране на елементите на низа. Във всеки един момент част от символите в буфера се използват, а останалите стоят в резерв. Това дава възможност добавянето да работи изключително бързо. Останалите операции също работят по-бързо, отколкото при класа **`string`**, защото промените не създават нов обект.

Нека създадем обект от класа **`StringBuilder`** с буфер от 15 символа. Към него ще добавим символния низ: "Hello,C#!". Получаваме следния код:

```
StringBuilder sb = new StringBuilder(15);
sb.Append("Hello,C#!");
```

След създаването на обекта и записването на стойността в него, той ще изглежда по следния начин:



Оцветените елементи са запълнената част от буфера с въведеното от нас съдържание. Обикновено при добавяне на нов символ към променливата не се създава нов обект в паметта, а се използва заделеното вече, но неизползвано пространство. Ако целият капацитет на буфера е запълнен, тогава вече се заделя нова област в динамичната памет с удвоен размер (текущия капацитет, умножен по 2) и данните се прехвърлят в нея. След това може отново да се добавят спокойно символи и символни низове, без нужда от непрекъснатото заделяне на памет.

StringBuilder – по-важни методи

Класът **`StringBuilder`** ни предоставя набор от методи, които ни помагат за лесно и ефективно редактиране на текстови данни и построяване на текст. Вече срещнахме някои от тях в примерите. По-важните са:

- **`StringBuilder(int capacity)`** – конструктор с параметър начален капацитет. Чрез него може предварително да зададем размера на буфера, ако имаме приблизителна информация за броя итерации и слепвания, които ще се извършат. Така спестяваме излишни заделяния на динамична памет.
- **`Capacity`** – връща размера на целия буфер (общият брой заети и свободни позиции в буфера).
- **`Length`** – връща дължината на записания низ в променливата (броя заети позиции в буфера).
- Индексатор **`[int index]`** – връща символа на указаната позиция.
- **`Append(…)`** – слепва низ, число или друга стойност след последния записан символ в буфера.

- **`Clear(…)`** – премахва всички символи от буфера (изтрива го).
- **`Remove(int startIndex, int length)`** – премахва (изтрива) низ от буфера по дадена начална позиция и дължина.
- **`Insert(int offset, string str)`** – вмъква низ на дадена позиция.
- **`Replace(string oldValue, string newValue)`** – замества всички срещания на даден подниз с друг подниз.
- **`ToString()`** – връща съдържанието на **`StringBuilder`** обекта във вид на **`string`**.

Извличане на главните букви от текст – пример

Следващата задача е да извлечем всички главни букви от даден текст. Можем да я реализираме по различни начини – използвайки масив и брояч и пълнейки масива с всички открити главни букви; създавайки обект от тип **`string`** и долепвайки главните букви една по една към него; използвайки класа **`StringBuilder`**.

Спирайки се на варианта за използване на масив, имаме проблем: не знаем какъв да бъде размерът на масива, тъй като предварително нямаме идея колко са главните букви в текста. Може да създадем масива толкова голям, колкото е текста, но по този начин хабим излишно място в паметта и освен това трябва да поддържаме брояч, който пази до къде е пълен масива.

Друг вариант е използването на променлива от тип **`string`**. Тъй като ще обходим целия текст и ще долепваме всички букви към променливата, вероятно е отново да загубим производителност заради конкатенирането на символни низове.

StringBuilder – правилното решение

Най-уместното решение на поставената задача отново е използването на **`StringBuilder`**. Можем да започнем с празен **`StringBuilder`**, да итерираме по буквите от зададения текст символ по символ, да проверяваме дали текущият символ от итерацията е главна буква и при положителен резултат да долепваме символа в края на нашия **`StringBuilder`**. Накрая можем да върнем натрупания резултат, който взимаме с извикването на метода **`ToString()`**. Следва примерна реализация:

```
public static string ExtractCapitals(string str)
{
  StringBuilder result = new StringBuilder();
  for (int i = 0; i < str.Length; i++)
  {
    char ch = str[i];
    if (char.IsUpper(ch))
    {
            result.Append(ch);
        }
    }
    return result.ToString();
}
```

Извиквайки метода **`ExtractCapitals(…)`** и подавайки му зададен текст като параметър, връщаната стойност е низ от всички главни букви в текста, т.е. началният низ с изтрити от него всички символи, които не са главни букви. За проверка дали даден символ е главна буква използваме **`char.IsUpper(…)`** – метод от стандартните класове в .NET. Можете да разгледате документацията за класа **`char`**, защото той предлага и други полезни методи за обработка на символи.

Форматиране на низове

.NET Framework предлага на програмиста механизми за форматиране на символни низове, числа и дати. Вече се запознахме с някои от тях в темата "Вход и изход от конзолата". Сега ще допълним знанията си с методите за форматиране и преобразуване на низове на класа **`string`**.

Служебният метод ToString(…)

Една от интересните концепции в .NET, е че практически всеки обект на клас, както и примитивните променливи, могат да бъдат представяни като текстово съдържание. Това се извършва чрез метода **`ToString(…)`**, който присъства във всички .NET обекти. Той е заложен в дефиницията на класа **`object`** – базовият клас, който наследяват пряко или непряко всички .NET типове данни. По този начин дефиницията на метода се появява във всеки един клас и можем да го ползваме, за да изведем във вид на някакъв текст съдържанието на всеки един обект.

Методът **`ToString(…)`** се извиква автоматично, когато извеждаме на конзолата обекти от различни класове. Например когато печатаме дати, скрито от нас подадената дата се преобразува до текст чрез извикване на **`ToString(…)`**:

```
DateTime currentDate = DateTime.Now;
Console.WriteLine(currentDate);
// 10.1.2010 г. 13:34:27 ч.
```

Когато подаваме **`currentDate`** като параметър на метода **`WriteLine(…)`**, нямаме точна декларация, която обработва дати. Методът има конкретна реализация за всички примитивни типове и символни низове. За всички останали обекти **`WriteLine(…)`** извиква метода им **`ToString(…)`**, който първо ги преобразува до текст, и след това извежда полученото текстово съдържание. Реално примерният код по-горе е еквивалентен на следния:

```
DateTime currentDate = DateTime.Now;
Console.WriteLine(currentDate.ToString());
```

Имплементацията по подразбиране на метода **ToString(…)** в класа **object** връща пълното име на съответния клас. Всички класове, които не предефинират изрично поведението на **ToString(…)**, използват именно тази имплементация. Повечето класове в C# имат собствена имплементация на метода, представяща четимо и разбираемо съдържанието на съответния обект във вид на текст. Например при преобразуване на число към текст се ползва стандартния за текущата култура формат на числата. При преобразуване на дата към текст също се ползва стандартния за текущата култура формат на датите.

Използване на String.Format(…)

String.Format(…) е статичен метод, чрез който можем да форматираме текст и други данни по шаблон (форматиращ низ). Шаблоните съдържат текст и декларирани параметри (placeholders) и служат за получаване на форматиран текст след заместване на параметрите от шаблона с конкретни стойности. Може да се направи директна асоциация с метода **Console.WriteLine(…)**, който също форматира низ по шаблон:

```
Console.WriteLine("This is a template from {0}", "Ivan");
```

Как да ползваме метода **String.Format(…)**? Нека разгледаме един пример, за да си изясним този въпрос:

```
DateTime date = DateTime.Now;
string name = "Svetlin Nakov";
string task = "Telerik Academy courses";
string location = "his office in Sofia";

string formattedText = String.Format(
   "Today is {0:dd.MM.yyyy} and {1} is working on {2} in {3}.",
   date, name, task, location);
Console.WriteLine(formattedText);

// Output: Today is 22.10.2010 and Svetlin Nakov is working on
// Telerik Academy courses in his office in Sofia.
```

Както се вижда от примера, форматирането чрез **String.Format()** използва параметри от вида **{0}**, **{1}**, и т.н. и приема форматиращи низове (като например **:dd.MM.yyyy**). Методът приема като първи параметър форматиращ низ, съдържащ текст с параметри, следван от стойностите за всеки от параметрите, а като резултат връща форматирания текст. Повече информация за форматиращите низове можете да намерите в Интернет и в статията Composite Formatting в MSDN (http://msdn.microsoft.com/en-us/library/txafckwd.aspx).

Парсване на данни

Обратната операция на форматирането на данни е тяхното парсване. **Парсване на данни** (**data parsing**) означава от текстово представяне на стойностите на някакъв тип в определен формат да се получи стойност от съответния тип, например от текста "22.10.2010" да се получи инстанция на типа `DateTime`, съдържаща съответната дата.

Често работата с приложения с графичен потребителски интерфейс предполага потребителският вход да бъде предаван през променливи от тип `string`, защото практически така може да се работи както с числа и символи, така и с текст и дати, форматирани по предпочитан от потребителя начин. Въпрос на опит на програмиста е да представи входните данни, които очаква, по правилния за потребителя начин. След това данните се преобразуват към по-конкретен тип и се обработват. Например числата могат да се преобразуват към променливи от `int` или `double`, а след това да участват в математически изрази за изчисления.

⚠ **При преобразуването на типове не бива да се осланяме само на доверието към потребителя. Винаги проверявайте коректността на входните потребителски данни! В противен случай ще настъпи изключение, което може да промени нормалната логика на програмата.**

Преобразуване към числови типове

За преобразуване на символен низ към число можем да използваме метода `Parse(…)` на примитивните типове. Нека видим пример за преобразуване на стринг към целочислена стойност (парсване):

```
string text = "53";
int intValue = int.Parse(text);
// intValue = 53
```

Можем да преобразуваме и променливи от булев тип:

```
string text = "True";
bool boolValue = bool.Parse(text);
// boolValue = true
```

Връщаната стойност е `true`, когато подаваният параметър е инициализиран (не е обект със стойност `null`) и съдържанието му е "`true`", без значение от регистъра на буквите в него, т.е. всякакви текстове като "true", "True" или "tRUe" ще зададат на променливата `boolValue` стойност `true`. Всички останали случаи връщат стойност `false`.

В случай, че подадената на `Parse(…)` метод стойност е невалидна за типа (например подаваме "Пешо" при преобразуване на число), се получава изключение.

Преобразуване към дата

Парсването към дата става по подобен начин, като парсването към числов тип, но е препоръчително да се зададе конкретен формат за датата. Ето един пример как може да стане това:

```
string text = "11.09.2001";
DateTime parsedDate = DateTime.Parse(text);
Console.WriteLine(parsedDate);
// 11-Sep-01 0:00:00 AM
```

Дали датата ще бъде успешно парсната и в какъв точно формат ще бъде отпечатана на конзолата зависи силно от текущата култура на Windows. В примера е използван модифициран вариант на американската култура (en-US). Ако искаме да зададем изрично формат, който не зависи от културата, можем да ползваме метода **`DateTime.ParseExact(…)`**:

```
string text = "11.09.2001";
string format = "dd.MM.yyyy";
DateTime parsedDate = DateTime.ParseExact(
  text, format, CultureInfo.InvariantCulture);
Console.WriteLine("Day: {0}\nMonth: {1}\nYear: {2}",
  parsedDate.Day, parsedDate.Month, parsedDate.Year);
// Day: 11
// Month: 9
// Year: 2001
```

При парсването по изрично зададен формат се изисква да се подаде конкретна култура, от която да се вземе информация за формата на датите и разделителите между дни и години. Тъй като искаме парсването да не зависи от конкретна култура, използваме неутралната култура: **`CultureInfo.InvariantCulture`**. За да използваме класа **`CultureInfo`**, трябва първо да включим пространството от имена **`System.Globalization`**.

Упражнения

1. Разкажете за низовете в C#. Какво е типично за типа **`string`**? Обяснете кои са най-важните методи на класа **`string`**.
2. Напишете програма, която прочита символен низ, обръща го отзад напред и го принтира на конзолата. Например: "**`introduction`**" → "**`noitcudortni`**".
3. Напишете програма, която проверява дали в даден аритметичен израз скобите са поставени коректно. Пример за израз с коректно поставени скоби: **`((a+b)/5-d)`**. Пример за некоректен израз: **`)(a+b))`**.
4. Колко обратни наклонени черти трябва да посочите като аргумент на метода **`Split(…)`**, за да разделите текста по обратна наклонена черта?

   Пример: **`one\two\three`**

Забележка: В C# обратната наклонена черта е екраниращ символ.

5. Напишете програма, която открива колко пъти даден подниз се съдържа в текст. Например нека търсим подниза "in" в текста:

```
We are living in a yellow submarine. We don't have anything
else. Inside the submarine is very tight. So we are drinking
all the day. We will move out of it in 5 days.
```

Резултатът е 9 срещания.

6. Даден е текст. Напишете програма, която променя регистъра на буквите до главни на всички места в текста, заградени с таговете **`<upcase>`** и **`</upcase>`**. Таговете не могат да бъдат вложени.

Пример:

```
We are living in a <upcase>yellow submarine</upcase>. We
don't have <upcase>anything</upcase> else.
```

Резултат:

```
We are living in a YELLOW SUBMARINE. We don't have ANYTHING
else.
```

7. Напишете програма, която чете от конзолата стринг от максимум 20 символа и ако е по-кратък го допълва отдясно със "**`*`**" до 20 символа.

8. Напишете програма, която преобразува даден стринг във вид на поредица от Unicode екраниращи последователности. Примерен входен стринг: "**`Наков`**". Резултат: "**`\u041d\u0430\u043a\u043e\u0432`**".

9. Напишете програма, която кодира текст по даден шифър като прилага шифъра побуквено с операция XOR (изключващо или) върху текста. Кодирането трябва да се извършва като се прилага XOR между първата буква от текста и първата буква на шифъра, втората буква от текста и втората буква от шифъра и т.н. до последната буква от шифъра, след което се продължава отново с първата буква от шифъра и поредната буква от текста. Отпечатайте резултата като поредица от Unicode кодирани екраниращи символи.

Примерен текст: "**`Nakov`**". Примерен шифър: "**`ab`**". Примерен резултат: "**`\u002f\u0003\u000a\u000d\u0017`**".

10. Напишете програма, която извлича от даден текст всички изречения, които съдържат определена дума. Считаме, че изреченията са разделени едно от друго със символа "`.`", а думите са разделени една от друга със символ, който не е буква. Примерен текст:

```
We are living in a yellow submarine. We don't have anything
```

```
else. Inside the submarine is very tight. So we are drinking
all the day. We will move out of it in 5 days.
```

Примерен резултат:

```
We are living in a yellow submarine.
We will move out of it in 5 days.
```

11. Даден е символен низ, съставен от няколко "забранени" думи, разделени със запетая. Даден е и текст, съдържащ тези думи. Да се напише програма, която замества забранените думи в текста със звездички. Примерен текст:

```
Microsoft announced its next generation C# compiler today. It
uses advanced parser and special optimizer for the Microsoft
CLR.
```

Примерен низ от забранените думи: "**`C#,CLR,Microsoft`**".

Примерен съответен резултат:

```
********* announced its next generation ** compiler today. It
uses advanced parser and special optimizer for the *********
***.
```

12. Напишете програма, която чете число от конзолата и го отпечатва в 15-символно поле, подравнено вдясно по няколко начина: като десетично число, като шестнайсетично число, като процент, като валутна сума и във вид на експоненциален запис (scientific notation).

13. Напишете програма, която приема URL адрес във формат:

```
[protocol]://[server]/[resource]
```

и извлича от него протокол, сървър и ресурс. Например при подаден адрес: **`http://www.devbg.org/forum/index.php`** резултатът е:

`[protocol]="http"`

`[server]="www.devbg.org"`

`[resource]="/forum/index.php"`

14. Напишете програма, която обръща думите в дадено изречение без да променя пунктуацията и интервалите. Например: "C# is not C++ and PHP is not Delphi" -> "Delphi not is PHP and C++ not is C#".

15. Даден е тълковен речник, който се състои от няколко реда текст. На всеки ред има дума и нейното обяснение, разделени с тире:

```
.NET – platform for applications from Microsoft
CLR – managed execution environment for .NET
```

```
namespace – hierarchical organization of classes
```

Напишете програма, която парсва речника и след това в цикъл чете дума от конзолата и дава обяснение за нея или съобщение, че думата липсва в речника.

16. Напишете програма, която заменя в HTML документ всички препратки (hyperlinks) от вида **`<a href="…">…</a>`** с препратки стил "форум", които имат вида **`[URL=…]…/URL]`**.

    Примерен текст:

```
<p>Please visit <a href="http://academy.telerik.com">our
site</a> to choose a training course. Also visit <a
href="www.devbg.org">our forum</a> to discuss the
courses.</p>
```

Примерен съответен резултат:

```
<p>Please visit [URL=http://academy.telerik.com]our
site[/URL] to choose a training course. Also visit
[URL=www.devbg.org]our forum[/URL] to discuss the
courses.</p>
```

17. Напишете програма, която чете две дати, въведени във формат "**`ден.месец.година`**" и изчислява броя дни между тях.

```
Enter the first date: 27.02.2006
Enter the second date: 3.03.2004
Distance: 4 days
```

18. Напишете програма, която чете дата и час, въведени във формат "**`ден.месец.година час:минути:секунди`**" и отпечатва датата и часа след 6 часа и 30 минути, в същия формат.

19. Напишете програма, която извлича от даден текст всички e-mail адреси. Това са всички поднизове, които са ограничени от двете страни с край на текст или разделител между думи и съответстват на формата **`<sender>@<host>…<domain>`**. Примерен текст:

```
Please contact us by phone (+359 222 222 222) or by email at
example@abv.bg or at baj.ivan@yahoo.co.uk. This is not email:
test@test. This also: @telerik.com. Neither this: a@a.b.
```

Извлечени e-mail адреси от примерния текст:

```
example@abv.bg
baj.ivan@yahoo.co.uk
```

20. Напишете програма, която извлича от даден текст всички дати, които се срещат изписани във формат **`DD.MM.YYYY`** и ги отпечатва на конзолата в стандартния формат за Канада. Примерен текст:

```
I was born at 14.06.1980. My sister was born at 3.7.1984. In
5/1999 I graduated my high school. The law says (see section
7.3.12) that we are allowed to do this (section 7.4.2.9).
```

    Извлечени дати от примерния текст:

```
14.06.1980
3.7.1984
```

21. Напишете програма, която извлича от даден текст всички думи, които са палиндроми, например "**`ABBA`**", "**`lamal`**", "**`exe`**".
22. Напишете програма, която чете от конзолата символен низ и отпечатва в азбучен ред всички букви от въведения низ и съответно колко пъти се среща всяка от тях.
23. Напишете програма, която чете от конзолата символен низ и отпечатва в азбучен ред всички думи от въведения низ и съответно колко пъти се среща всяка от тях.
24. Напишете програма, която чете от конзолата символен низ и заменя в него всяка последователност от еднакви букви с единична съответна буква. Пример: "**aaaaabbbbbcdddeeeedssaa**" → "**abcdedsa**".
25. Напишете програма, която чете от конзолата списък от думи, разделени със запетайки и ги отпечатва по азбучен ред.
26. Напишете програма, която изважда от даден HTML документ всичкия текст без таговете и техните атрибути.

    Примерен текст:

```
<html>
  <head><title>News</title></head>
  <body><p><a href="http://academy.telerik.com">Telerik
    Academy</a>aims to provide free real-world practical
    training for young people who want to turn into
    skillful .NET software engineers.</p></body>
</html>
```

    Примерен съответен резултат:

```
Title: News
Body:
Telerik Academy aims to provide free real-world practical
training for young people who want to turn into skillful .NET
software engineers.
```

Решения и упътвания

1. Прочетете в MSDN или вижте първия абзац в тази глава.
2. Използвайте `StringBuilder` и `for` (или `foreach`) цикъл.
3. Използвайте броене на скобите: при отваряща на скоба увеличавайте брояча с 1, а при затваряща го намалявайте с 1. Следете броячът да не става отрицателно число и да завършва винаги на 0.
4. Ако не знаете колко наклонени черти трябва да използвате, изпробвайте `Split(…)` с нарастващ брой черти, докато достигнете до желания резултат.
5. Обърнете регистъра на буквите в текста до малки и търсете в цикъл дадения подниз. Не забравяйте да използвате `IndexOf(…)` с начален индекс, за да избегнете безкраен цикъл.
6. Използвайте регулярни изрази или `IndexOf(…)` за отварящ и затварящ таг. Пресметнете началния и крайния индекс на текста. Обърнете текста в главни букви и заменете целия подниз **отварящ таг + текст + затварящ таг** с текста в горен регистър.
7. Използвайте метода `PadRight(…)` от класа `String`.
8. Използвайте форматиращ низ форматиращ низ "`\u{0:x4}`" за Unicode кода на всеки символ от входния стринг (можете да го получите чрез преобразуване на `char` към `ushort`).
9. Нека шифърът `cipher` се състои от `cipher.Length` букви. Завъртете цикъл по буквите от текста и буквата на позиция `index` в текста шифрирайте с `cipher[cipher.Length % index]`. Ако имаме буква от текста и буква от шифъра, можем да извършим XOR операция между тях като предварително превърнем двете букви в числа от тип `ushort`. Можем да отпечатаме резултата с форматиращ низ "`\u{0:x4}`".
10. Първо разделете изреченията едно от друго чрез метода `Split(…)`. След това проверявайте дали всяко от изреченията съдържа търсената дума като я търсите като подниз с `IndexOf(…)` и ако я намерите проверявате дали отляво и отдясно на намерения подниз има разделител (символ, който не е буква или начало / край на низ).
11. Първо разделете забранените думи с метода `Split(…)`, за да ги получите като масив. За всяка забранена дума обхождайте текста и търсете срещане. При срещане на забранена дума, заменете с толкова звездички, колкото букви се съдържат в забранената дума.

    Друг, по-лесен, подход е да използвате `RegEx.Replace(…)` с подходящ регулярен израз и подходящ `MatchEvaluator` метод.
12. Използвайте подходящи форматиращи низове.

13. Използвайте регулярен израз или търсете по съответните разделители – две наклонени черти за край на протокол и една наклонена черта за разделител между сървър и ресурс. Разгледайте специалните случаи, в които части от URL адреса могат да липсват.

14. Можете да решите задачата на две стъпки: обръщане на входния низ на обратно; обръщане на всяка от думите от резултата на обратно.

    Друг, интересен подход е да разделете входния текст по препинателните знаци между думите, за да получите само думите от текста и след това да разделите по буквите, за да получите препинателните знаци от текста. Така, имайки списък от думи и списък от препинателни знаци между тях, лесно можете да обърнете думите на обратно, запазвайки препинателните знаци.

15. Можете да парснете текст като го разделите първо по символа на нов ред, а след това втори път по " - ". Речникът е най-удачно да запишете във хеш-таблица (`Dictionary<string, string>`), която ще осигури бързо търсене по зададена дума. Прочетете в Интернет за хеш-таблици и за класа `Dictionary<K,T>`.

16. Най-лесно задачата може да решите с регулярен израз.

    Ако все пак изберете да не ползвате регулярни изрази, може да намерите всички поднизове, които започват с "`<a href=`" и завършват с "`</a>`" и вътре в тях да замените "`<a href="`" с "`[URL=`", след това първото срещнато "`">`" с "`]`" и след това "`</a>`" с "`[/URL]`".

17. Използвайте методите на структурата `DateTime`, а за парсване на датите може да ползвате разделяне по "." или парсване с метода `DateTime.ParseExact(…)`.

18. Използвайте методите `DateTime.ToString()` и `DateTime.ParseExact()` с подходящи форматиращи низове.

19. Използвайте `RegEx.Match(…)` с подходящ регулярен израз.

    Ако решавате задачата без регулярни изрази, ще трябва да обработвате текста побуквено от начало до край и да обработвате поредния символ в зависимост от текущия режим на работа, който може да е един `OutsideOfEmail`, `ProcessingSender` или `ProcessingHostOrDomain`. При срещане на разделител или край на текста, ако се обработва хост или домейн (режим `ProcessingHostOrDomain`), значи е намерен email, а иначе потенциално започва нов e-mail и трябва да се премине в състояние `ProcessingSender`. При срещане на `@` в режим на работа `ProcessingSender` се преминава към режим `ProcessingSender`. При срещане на букви или точка в режими `ProcessingSender` или `ProcessingHostOrDomain` те се натрупват в буфер. По пътя на тези разсъждения можете да разглеждате всички възможни групи символи, срещнати съответно във всеки от трите режима и да ги обработите по подходящ начин. Реално се получава нещо като краен автомат (state machine), който разпознава e-mail адреси. Всички намерени e-mail адреси трябва да се проверят дали имат непразен получател, непразен хост, домейн с дължина между 2 и 4 букви, както и да не започват или завършват с точка.

Друг по-лесен подход за тази задача е да се раздели текста по всички символи, които не са букви и точки и да се проверят така извлечените "думи" дали са валидни e-mail адреси чрез опит да се раздробят на непразни части: **`<sender>`**, **`<host>`**, **`<domain>`**, отговарящи на изброените вече условия.

20. Използвайте **`RegEx.Match(…)`** с подходящ регулярен израз. Алтернативният вариант е да си реализирате автомат, който има състояния **`OutOfDate`**, **`ProcessingDay`**, **`ProcessingMonth`**, **`ProcessingYear`** и обработвайки текста побуквено да преминавате между състоянията според поредната буква, която обработвате. Както и при предходната задача, можете предварително да извадите всички "думи" от текста и след това да проверите кои от тях съответстват на шаблона за дата.

21. Раздробете текста на думи и проверете всяка от тях дали е палиндром.

22. Използвайте масив от символи **`char[65536]`**, в който ще отбелязвате колко пъти се среща всяка буква. Първоначално всички елементи на масива са нули. След побуквена обработка на входния низ можете да отбележите в масива коя буква колко пъти се среща. Примерно ако се срещне буквата '**`A`**', ще се увеличи с единици броят срещания в масива на индекс 65 (Unicode кодът на '**`A`**'). Накрая с едно сканиране на масива може да се отпечатат всички ненулеви елементи (като се преобразуват **`char`**, за да се получи съответната буква) и и прилежащия им брой срещания.

23. Използвайте хеш-таблица (**`Dictionary<string,int>`**), в която пазите за всяка дума от входния низ колко пъти се среща. Прочетете в Интернет за класа **`System.Collections.Generic.Dictionary<K,T>`**. С едно обхождане на думите можете да натрупате в хеш-таблицата информация за срещанията на всяка дума, а с обхождане на хеш-таблицата можете да отпечатате резултата.

24. Можете да сканирате текста отляво надясно и когато текущата буква съвпада с предходната, да я пропускате, а в противен случай да я долепяте в **`StringBuilder`**.

25. Използвайте статичния метод **`Array.Sort(…)`**.

26. Сканирайте текста побуквено и във всеки един момент пазете в една променлива дали към момента има отворен таг, който не е бил затворен или не. Ако срещнете "<", влизайте в режим "отворен таг". Ако срещнете ">", излизайте от режим "отворен таг". Ако срещнете

буква, я добавяйте към резултата, само ако програмата не е в режим "отворен таг". След затваряне на таг може да добавяте по един интервал, за да не се слепва текст преди и след тага.

Telerik Academy SCHOOL

Стани част от Академията на Телерик за ученици!

"Основната цел на обучението е да израстнеш, а нашите умове, за разлика от телата ни, могат да растат през целия ни живот!" Мортимър Адлър

АКАДЕМИЯТА НА ТЕЛЕРИК за ученици предоставя съвременно, специализирано обучение по софтуерно инженерство за ученици от средните училища в България. Нашата основна цел е да помогнем на младите момчета и момичета да се подготвят за Националната олимпиада по информационни технологии (НОИТ) и да придобият солидни теоретични и практически умения за разработка на софтуер и софтуерни технологии.

В Училищната академия на Телерик по софтуерно инженерство:

- **Стимулираме креативността и мисленето на учениците**
- **Всеки ученик получава индивидуално внимание и съдействие**
- **Полагаме основите за писане на качествен програмен код**
- **Изучаваме съвременните софтуерни технологии**
- **Подготвяме учениците за Националната олимпиада по информационни технологии**
- **Предлагаме голямо разнообразие от практически и теоретични обучения**

Програмата на Академията включва:

C#, .NET технологии, бази данни, SQL, Silverlight, 3D графики и моделиране, разработка на игри, програмиране на микроконтролери, WPF, ASP.NET, HTML5, разработка на мобилни приложения, основи на софтуерното инженерство, работа в екип

schoolacademy.telerik.com

groups.google.com/group/it-olymp *facebook.com/TelerikSchoolAcademy*

telerik
deliver more than expected

Глава 14. Дефиниране на класове

В тази тема...

В настоящата тема ще разберем как можем да дефинираме собствени класове и кои са елементите на класовете. Ще се научим да декларираме полета, конструктори и свойства в класовете. Ще припомним какво е метод и ще разширим знанията си за модификатори и нива на достъп до полетата и методите на класовете. Ще разгледаме особеностите на конструкторите и подробно ще обясним как обектите се съхраняват в динамичната памет и как се инициализират полетата им. Накрая ще обясним какво представляват статичните елементи на класа – полета (включително константи), свойства и методи и как да ги ползваме.

Собствени класове

"*... Всеки модел представя някакъв аспект от реалността или някаква интересна идея. Моделът е опростяване. Той интерпретира реалността, като се фокусира върху аспектите от нея, свързани с решаването на проблема и игнорира излишните детайли.*" [Evans]

Целта на всяка една програма, която създаваме, е да реши даден проблем или да реализира някаква идея. За да измислим решението, ние първо създаваме опростен модел на реалността, който не отразява всички факти от нея, а се фокусира само върху тези, които имат значение за намирането на решение на нашата задача. След това, използвайки модела, намираме решение (т.е. създаваме алгоритъма) на нашия проблем и това решение го описваме чрез средствата на даден език за програмиране.

В днешно време най-често използваният тип езици за програмиране са обектно-ориентираните. И тъй като обектно-ориентираното програмиране (ООП) е близко до начина на мислене на човека, то ни дава възможността с лекота да описваме модели на заобикалящата ни среда. Една от причините за това е, че ООП ни предоставя средство, за описание на съвкупността от понятия, които описват обектите във всеки модел. Това средство се нарича клас (class). Понятието клас и дефинирането на собствени класове, различни от системните, е вградена възможност на езика C# и целта на настоящата глава е да се запознаем с него.

Да си припомним: какво са класовете и обектите?

Клас (**class**) в ООП наричаме описание (спецификация) на даден клас обекти от реалността. Класът представлява шаблон, който описва видовете състояния и поведението на конкретните обекти (екземплярите), които биват създавани от този клас (шаблон).

Обект (**object**) наричаме екземпляр създаден по дефиницията (описанието) на даден клас. Когато един обект е създаден по описанието, което един клас дефинира, казваме, че **обектът е от тип "името на този клас"**.

Например, ако имаме клас `Dog`, описващ някакви характеристики на куче от реалния свят, казваме, че обектите, които са създадени по описанието на този клас (например кученцата "Шаро" и "Рекс") са от тип класа `Dog`. Това означение е същото, както когато казваме, че низът "some string" е от класа `String`. Разликата е, че обектът от тип `Dog` е екземпляр от клас, който не е част от библиотеката с класове на .NET Framework, а е дефиниран от самите нас.

Какво съдържа един клас?

Всеки клас съдържа дефиниция на това какви данни трябва да се съдържат в един обект, за да се опише състоянието му. Обектът (конкретния екземпляр от този клас) съдържа самите данни. Тези данни дефинират състоянието му.

Освен състоянието, в класа също се описва и поведението на обектите. Поведението се изразява в действията, които могат да бъдат извършвани от обектите. Средството на ООП, чрез което можем да описваме поведението на обектите от даден клас, е декларирането на методи в класа.

Елементи на класа

Сега ще изброим основните елементи на един клас, а по-късно ще разгледаме подробно всеки един от тях.

Основните елементи на класовете в C# са следните:

- **Декларация на класа (class declaration)** – това е редът, на който декларираме името на класа. Например:

```
public class Dog
```

- **Тяло на клас** – по подобие на методите, класовете също имат част, която следва декларацията им, оградена с фигурни скоби – "`{`" и "`}`" между които се намира съдържанието на класа. Тя се нарича тяло на класа. Елементите на класа, които се описват в тялото му са изброени в следващите точки.

```
public class Dog
{
  // ... The body of the class comes here ...
}
```

- **Конструктор (constructor)** – това е псевдометод, който се използва за създаване на нови обекти. Така изглежда един конструктор:

```
public Dog()
{
  // ... Some code ...
}
```

- **Полета (fields)** – те са променливи, декларирани в класа (някъде в литературата се срещат като **член-променливи**). В тях се пазят данни, които отразяват състоянието на обекта и са нужни за работата на методите на класа. Стойността, която се пази в полетата, отразява конкретното състояние на дадения обект, но съществуват и такива полета, наречени **статични**, които са общи за всички обекти.

```
// Field definition
private string name;
```

- **Свойства (properties)** – така наричаме характеристиките на даден клас. Обикновено стойността на тези характеристики се пази в полета. Подобно на полетата, свойствата могат да бъдат притежавани само от конкретен обект или да са споделени между всички обекти от тип даден клас.

```
// Property definition
private string Name { get; set; }
```

- **Методи (methods)** – от главата "Методи", знаем, че методите представляват именувани блокове програмен код. Те извършват някакви действия и чрез тях реализират поведението на обектите от този клас. В методите се изпълняват алгоритмите и се обработват данните на обекта.

Ето как изглежда един клас, който сме дефинирали сами и който притежава елементите, които описахме току-що:

```
// Class declaration
public class Dog
{  // Opening brace of the class body

   // Field declaration
   private string name;

   // Constructor declaration
   public Dog()
   {
      this.name = "Balkan";
   }

   // Another constructor declaration
   public Dog(string name)
   {
      this.name = name;
   }

   // Property declaration
   public string Name
   {
      get { return name; }
      set { name = value; }
   }

   // Method declaration
   public void Bark()
   {
      Console.WriteLine("{0} said: Wow-wow!", name);
   }
}  // Closing brace of the class body
```

За момента няма да обясняваме в по-големи детайли изложения код, тъй като подробна информация ще бъде дадена при обяснението как се декларира всеки един от елементите на класа.

Използване на класове и обекти

В главата "Създаване и използване на обекти" видяхме подробно как се създават нови обекти от даден клас и как могат да се използват. Сега накратко ще си припомним как ставаше това.

Как да използваме дефиниран от нас клас?

За да можем да използваме някой клас, първо трябва да създадем обект от него. За целта използваме ключовата дума **`new`** в комбинация с някой от конструкторите на класа. Това ще създаде обект от дадения клас (тип).

За да можем да манипулираме новосъздадения обект, ще трябва да го присвоим на променлива от типа на неговия клас. По този начин в тази променлива ще бъде запазена връзка (референция) към него.

Чрез променливата, използвайки точкова нотация, можем да извикваме методите, свойствата на обекта, както и да достъпваме полетата (член-променливите) и свойствата му.

Пример – кучешка среща

Нека вземем примера от предходната секция на тази глава, където дефинирахме класа **`Dog`**, който описва куче, и добавим метод **`Main()`** към него. В него ще онагледим казаното току-що:

```
static void Main()
{
   string firstDogName = null;
   Console.WriteLine("Write first dog name: ");
   firstDogName = Console.ReadLine();

   // Using a constructor to create a dog with specified name
   Dog firstDog = new Dog(firstDogName);

   // Using a constructor to create a dog wit a default name
   Dog secondDog = new Dog();

   Console.WriteLine("Write second dog name: ");
   string secondDogName = Console.ReadLine();

   // Using property to set the name of the dog
   secondDog.Name = secondDogName;

   // Creating a dog with a default name
   Dog thirdDog = new Dog();

   Dog[] dogs = new Dog[] { firstDog, secondDog, thirdDog };

   foreach (Dog dog in dogs)
   {
      dog.Bark();
   }
}
```

Съответно изходът от изпълнението ще бъде следният:

```
Write first dog name:
Bobcho
Write second dog name:
Walcho
Bobcho said: Wow-wow!
Walcho said: Wow-wow!
Balkan said: Wow-wow!
```

В примерната програма, с помощта на **Console.ReadLine()**, получаваме имената на обектите от тип куче, които потребителят трябва да въведе от конзолата.

Присвояваме първия въведен низ на променливата **firstDogName**. След това използваме тази променлива при създаването на първия обект от тип **Dog** – **firstDog**, като я подаваме като параметър на конструктора.

Създаваме втория обект от тип **Dog**, без да подаваме низ за името на кучето на конструктора му. След това, чрез **Console.ReadLine()**, въвеждаме името на второто куче и получената стойност директно подаваме на свойството **Name**. Извикването му става чрез точкова нотация, приложена към променливата, която пази референция към втория създаден обект от тип **Dog** – **secondDog.Name**.

Когато създаваме третия обект от тип **Dog**, не подаваме име на кучето на конструктора, нито след това модифицираме подразбиращата се стойност "**Balkan**".

След това създаваме масив от тип **Dog**, като го инициализираме с трите обекта, които току-що създадохме.

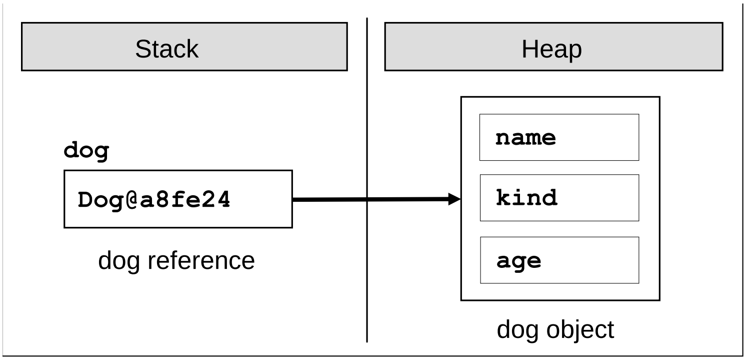
Накрая, използваме цикъл, за да обходим масива от обекти от тип **Dog**. На всеки елемент от масива, отново използвайки точкова нотация, извикваме метода **Bark()** за съответния обект чрез **dog.Bark()**.

Природа на обектите

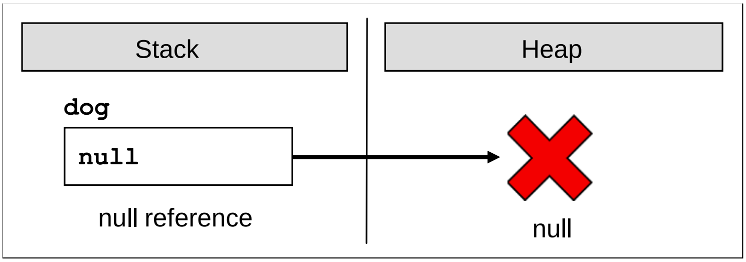
Нека припомним, че когато в .NET създадем един обект, той се състои от две части – същинска част от обекта, която съдържа неговите данни и се намира в частта от оперативната памет, наречена динамична памет (heap) и референция към този обект, която се намира в друга част от оперативната памет, където се държат локалните променливи и параметрите на методите, наречена стек (stack).

Например, нека имаме клас **`Dog`**, на който характеристиките му са име (name), порода (kind) и възраст (age). Създаваме променлива **`dog`** от този клас. Тази променлива се явява референция (указател) към обекта в динамичната памет (heap).

Референцията е променливата, чрез която достъпваме обекта. На схемата по-долу примерната референция, която има връзка към реалния обект в хийпа, е с името **`dog`**. В нея, за разлика от променливите от примитивен тип, не се съдържа самата стойност (т.е. данните на самия обект), а адреса, на който те се намират в хийпа:



Когато декларираме една променлива от тип някакъв клас, но не искаме тя да е инициализирана с връзка към конкретен обект, тогава трябва да ѝ присвоим стойност **`null`**. Ключовата дума **`null`** в езика C# означава, че една променлива не сочи към нито един обект (липса на стойност):



Съхранение на собствени класове

В C# единственото ограничение относно съхранението на наши собствени класове е те да са във файлове с разширение **`.cs`**. В един такъв файл може да има няколко класа, структури и други типове. Въпреки че компилаторът не го изисква, е препоръчително всеки клас да се съхранява в отделен файл, който съответства на името му, т.е. класът **`Dog`** трябва да е записан във файл с име **`Dog.cs`**.

Вътрешна организация на класовете

Както знаем от темата "Създаване и използване на обекти", пространствата от имена (namespaces) в C# представляват именувани групи класове, които са логически свързани, без да има специално изискване как да бъдат разположени във файловата система.

Ако искаме да включим в кода си пространствата от имена нужни за работата на класовете, декларирани в даден файл или няколко файла, това трябва да стане чрез т.нар. **`using`** директиви. Те не са задължителни, но ако ги има, трябва да ги поставим на първите редове от файла, преди декларациите на класове или други типове. В следващите параграфи ще разберем за какво по-точно служат те.

След включването на използваните пространства от имена, следва декларирането на пространството от имена на класовете във файла. Както вече знаем, не сме задължени да дефинираме класовете си в пространство от имена, но е добра практика да го правим, тъй като разпределянето в пространства от имена помага за по-добрата организация на кода и разграничаването на класовете с еднакви имена.

Пространствата от имена съдържат декларации на класове, структури, интерфейси и други типове данни, както и други пространства от имена. Пример за вложени пространства от имена е пространството от имена **`System`**, което съдържа пространството от имена **`Data`**. Името на вложеното пространство е **`System.Data`**.

Пълното име на класа в .NET Framework е името на класа, предшествано от името на пространството от имена, в което той е деклариран: **`<namespace_name>.<class_name>`**. Чрез **`using`** директивите можем да използваме типовете от дадено пространство от имена, без да уточняваме пълното му име. Например:

```
using System;
…
DateTime date;
```

вместо

```
System.DateTime date;
```

Ето типичната последователност на декларациите, която трябва да следваме, когато създаваме собствени **`.cs`** файлове:

```
// Using directives - optional
using <namespace1>;
using <namespace2>;

// Namespace definition - optional
namespace <namespace_name>
```

```
{
    // Class declaration
    class <first_class_name>
    {
        // ... Class body ...
    }

    // Class declaration
    class <second_class_name>
    {
        // ... Class body ...
    }

    // ...

    // Class declaration
    class <n-th_class_name>
    {
        // ... Class body ...
    }
}
```

Декларирането на пространство от имена и съответно включването на пространства от имена са вече обяснени в главата "Създаване и използване на обекти" и затова няма да ги дискутираме отново.

Преди да продължим, да обърнем внимание на първия ред от горната схема. Вместо включвания на пространства от имена той съдържа коментар. Това не е проблем, тъй като по време на компилация, коментарите се "изчистват" от кода и на първи ред от файла остава включване на пространство от имена.

Кодиране на файловете. Четене на кирилица и Unicode

Когато създаваме `.cs` файл, в който да дефинираме класовете си, е добре да помислим за кодирането при съхраняването му във файловата система.

Вътрешно в .NET Framework компилираният код се представя в Unicode кодиране и затова няма проблеми, ако във файла използваме символи, които са от азбуки, различни от латинската, например на кирилица:

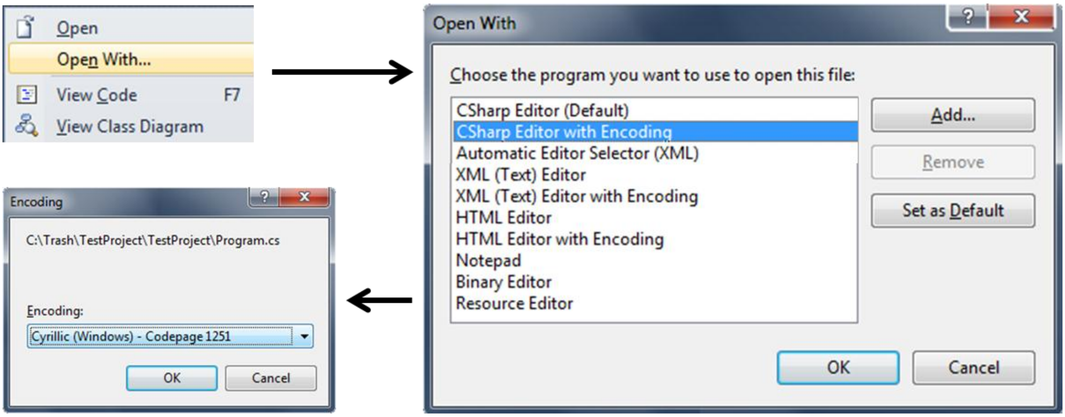
```
using System;

public class EncodingTest
{
    // Тестов коментар
    static int години = 4;

    static void Main()
```

```
  {
    Console.WriteLine("години: " + години);
  }
}
```

Този код ще се компилира и изпълни без проблем, но за да запазим символите четими в редактора на Visual Studio, трябва да осигурим подходящото кодиране на файла.

За да направим това или ако искаме да използваме различно кодиране от Unicode, трябва да асоциираме съответното кодиране с файла. При отваряне на файлове това става по следния начин:

1. От **File** менюто избираме **Open** и след това **File**.
2. В прозореца **Open File** натискаме стрелката, съседна на бутона **Open** и избираме **Open With**.
3. От списъка на прозореца **Open With** избираме **Editor** с **encoding support**, например **CSharp Editor with Encoding**.
4. Натискаме **[OK]**.
5. В прозореца **Encoding** избираме съответното кодиране от падащото меню **Encoding**.
6. Натискаме **[OK]**.


Open
Open With...
View Code F7
View Class Diagram
Open With
Choose the program you want to use to open this file:
CSharp Editor (Default)
CSharp Editor with Encoding
Automatic Editor Selector (XML)
XML (Text) Editor
XML (Text) Editor with Encoding
HTML Editor
HTML Editor with Encoding
Notepad
Binary Editor
Resource Editor
Add...
Remove
Set as Default
OK
Cancel
Encoding
OK
Cancel

C:\Trash\TestProject\TestProject\Program.cs

Encoding: Cyrillic (Windows) - Codepage 1251

За запаметяване на файлове във файловата система в определено кодиране стъпките са следните:

1. От менюто **File** избираме **Save As**.
2. В прозореца **Save File As** натискаме стрелката, съседна на бутона **Save** и избираме **Save with Encoding**.
3. В **Advanced Save Options** избираме желаното кодиране от списъка **Encoding** (за предпочитане е универсалното кодиране UTF-8).
4. От **Line Endings** избираме желания вид за край на реда.

Въпреки че имаме възможността да използваме символи от други азбуки, в `.cs` файловете, е препоръчително да пишем всички идентификатори и коментари на английски език, за да може кодът ни да е разбираем за повече хора по света.

Представете си само, ако ви се наложи да дописвате код, писан от виетнамец, където имената на променливите и коментарите са на виетнамски език. Не искате да ви се случва, нали? Тогава се замислете как ще се почувства един виетнамец, ако види променливи и коментари на български език.

Модификатори и нива на достъп (видимост)

Нека си припомним, от главата "Методи", че модификатор наричаме ключова дума с помощта, на която даваме допълнителна информация на компилатора за кода, за който се отнася модификаторът.

В C# има четири модификатора за достъп. Те са **`public`**, **`private`**, **`protected`** и **`internal`**. Модификатори за достъп могат да се използват само пред следните елементи на класа: декларация, полета, свойства и методи на класа.

Модификатори и нива на достъп

Както обяснихме, в C# има четири модификатора за достъп – **`public`**, **`private`**, **`protected`** и **`internal`**. С тях ние ограничаваме или позволяваме достъпа (видимостта) до елементите на класа, пред които те са поставени. Нивата на достъп в .NET биват **`public`**, **`protected`**, **`internal`**, **`protected internal`** и **`private`**. В тази глава ще се занимаем подробно само с **`public`**, **`private`** и **`internal`**. Повече за **`protected`** и **`protected internal`** ще научим в главата "Принципи на обектно-ориентираното програмиране".

Ниво на достъп public

Използвайки модификатора **`public`**, ние указваме на компилатора, че елементът, пред който е поставен, може да бъде достъпен от всеки друг клас, независимо дали е от текущия проект, от текущото пространство от имена или извън тях. Нивото на достъп **`public`** определя липса на ограничения върху видимостта, най-малко рестриктивното от всички нива на достъп в C#.

Ниво на достъп private

Нивото на достъп **`private`**, което налага най-голяма рестрикция на видимостта на класа и елементите му. Модификаторът **`private`** служи за индикация, че елементът, за който се отнася, не може да бъде достъпван от никой друг клас (освен от класа, в който е дефиниран), дори този клас да се намира в същото пространство от имена. Това ниво на достъп се използва по подразбиране, т.е. се прилага, когато липсва модификатор за достъп пред съответния елемент на класа.

Ниво на достъп internal

Модификаторът `internal` се използва, за да се ограничи достъпът до елемента само от файлове от същото асембли, т.е. същия проект във Visual Studio. Когато във Visual Studio направим няколко проекта, класовете от тях ще се компилират в различни асемблита.

Асембли (assembly)

Асембли (assembly) е колекция от типове и ресурси, която формира логическа единица функционалност. Всички типове в C# и изобщо в .NET Framework могат да съществуват само в асемблита. При всяка компилация на .NET приложение се създава асембли. То се съхранява като файл с разширение `.exe` или `.dll`.

Деклариране на класове

Декларирането на клас следва строго определени правила (синтаксис):

```
[<access_modifier>] class <class_name>
```

Когато декларираме клас, задължително трябва да използваме ключовата дума `class`. След нея трябва да стои името на класа `<class_name>`.

Освен ключовата дума `class` и името на класа, в декларацията на класа могат да бъдат използвани някои модификатори, например разгледаните вече модификатори за достъп.

Видимост на класа

Нека имаме два класа – `A` и `B`. Казваме, че класът `A`, има достъп до класа `B`, ако може да прави едно от следните неща:

1. Създава обект (инстанция) от тип класа `B`.
1. Достъпва определени методи и член-променливи (полета) в класа `B`, в зависимост от нивото на достъп на съответните методи и полета.

Има и трета операция, която може да бъде извършвана с класове, когато видимостта им позволява, наречена **наследяване на клас**, но на нея ще се спрем по-късно, в главата "Принципи на обектно-ориентираното програмиране".

Както разбрахме, ниво достъп означава "видимост". Ако класът `A` не може да "види" класа `B`, нивото на достъп на методите и полетата в класа `B` нямат значение.

Нивата на достъп, които един невложен клас може да има, са само `public` и `internal`.

Ниво на достъп public

Ако декларираме един клас с модификатор за достъп `public`, ще можем да го достъпваме от всеки един клас и от всичко едно пространство от имена, независимо къде се намират те. Това означава, че всеки друг клас ще може да създава обекти от тип този клас и да има достъп до методите и полетата на класа (стига тези полета да имат подходящо ниво на достъп).

Не трябва да забравяме, че ако искаме да използваме клас с ниво на достъп `public` от друго пространство от имена, различно от текущото, трябва да използваме конструкцията за включване на пространства от имена `using` или всеки път да изписваме пълното име на класа.

Ниво на достъп internal

Ако декларираме един клас с модификатор за достъп `internal`, той ще бъде достъпен само от същото асембли. Това означава, че само класовете от същото асембли ще могат да създават обекти от тип този клас и да имат достъп до методите и полетата (с подходящо ниво на достъп) на класа. Това ниво на достъп се подразбира, когато не е използван никакъв модификатор за достъп при декларацията на класа.

Ако във Visual Studio имаме два проекта в общ solution и искаме от единия проект да използваме клас, дефиниран в другия проект, то реферираният клас трябва задължително да е `public`.

Ниво на достъп private

За да сме изчерпателни, трябва да споменем, че като модификатор за достъп до клас, може да се използва модификаторът за видимост `private`, но това е свързано с понятието "вътрешен клас" (nested class), което ще разгледаме в секцията "Вътрешни класове".

Тяло на класа

По подобие на методите, след декларацията на класа следва неговото тяло, т.е. частта от класа, в която се съдържа програмния код:

```
[<access_modifier>] class <class_name>
{
   // ... Class body – the code of the class goes here ...
}
```

Тялото на класа започва с отваряща фигурна скоба "`{`" и завършва със затваряща – "`}`". Класът винаги трябва да има тяло.

Правила за именуване на класовете

По подобие на декларирането на име на метод, за създаването на име на клас съществува следния общоприет стандарт:

1. Имената на класовете започват с главна буква, а останалите букви са малки. Ако името е съставено от няколко думи, всяка дума започва с главна буква, без да се използват разделители между думите.
2. За имена на класове обикновено се използват съществителни имена.
3. Името на класовете е препоръчително да бъде на английски език.

Ето няколко примера за имена на класове, които са правилно именувани:

```
Dog
Account
Car
BufferedReader
```

Повече за имената на класовете ще научите в главата "Качествен програмен код".

Ключовата дума this

Ключовата дума **`this`** в C# дава достъп до референцията към текущия обект, когато се използва от метод в даден клас. Това е обектът, чийто метод или конструктор бива извикван. Можем да я разглеждаме като указател (референция), дадена ни априори от създателите на езика, с която да достъпваме елементите (полета, методи, конструктори) на собствения ни клас:

```
this.myField; // access a field in the class
this.DoMyMethod(); // access a method in the class
this(3, 4); // access a constructor with two int parameters
```

За момента няма да обясняваме изложения код. Разяснения ще дадем по-късно, в местата от секциите на тази глава, посветени на елементите на класа (полета, методи, конструктори) и засягащи ключовата дума **`this`**.

Полета

Както стана дума в началото на главата, когато декларираме клас, описваме обект от реалния свят. За описанието на този обект, се фокусираме само върху характеристиките му, които имат отношение към проблема, който ще решава нашата програма.

Тези характеристики на реалния обект ги интерпретираме в декларацията на класа, като декларираме набор от специален тип променливи, наречени полета, в които пазим данните за отделните характеристики. Когато създадем обект по описанието на нашия клас, стойностите на полетата, ще съдържат конкретните характеристики, с които даден екземпляр от класа (обект) се отличава от всички останали обекти от същия клас.

Деклариране на полета в даден клас

До момента сме се сблъсквали само с два типа променливи (вж. главата "Методи"), в зависимост от това къде са декларирани:

1. **Локални променливи** – това са променливите, които са дефинирани в тялото на някой метод (или блок).
2. **Параметри** – това са променливите в списъка с параметри, които един метод може да има.

В C# съществува и трети вид променливи, наречени **полета** (**fields**) или **член-променливи на класа** (**instance variables**).

Те се декларират в тялото на класа, но извън тялото на блок, метод или конструктор (какво е конструктор ще разгледаме подробно след малко).

Полетата се декларират в тялото на класа, но извън тялото на метод, конструктор или блок.

Ето един примерен код, в който се декларират няколко полета:

```
class SampleClass
{
   int age;
   long distance;
   string[] names;
   Dog myDog;
}
```

Формално, декларацията на полетата става по следния начин:

```
[<modifiers>] <field_type> <field_name>;
```

Частта **`<field_type>`** определя типа на даденото поле. Той може да бъде както примитивен тип (**`byte`**, **`short`**, **`char`** и т.н.) или масив, така и от тип някакъв клас (например **`Dog`** или **`string`**).

Частта **`<field_name>`** е името на даденото поле. Както при имената на обикновените променливи, когато именуваме една член-променлива, трябва да спазваме правилата за именуване на идентификатори в C# (вж. главата "Примитивни типове и променливи").

Частта **`<modifiers>`** е понятие, с което сме означили както модификаторите за достъп, така и други модификатори. Те не са задължителна част от декларацията на едно поле.

Модификаторите и нивата на достъп, позволени в декларацията на едно поле, са обяснени в секцията "Видимост на полета и методи".

В тази глава, от другите модификатори, които не са за достъп, и могат да се използват при декларирането на полета на класа, ще обърнем внимание още на **`static`**, **`const`** и **`readonly`**.

Област на действие (scope)

Трябва да знаем, че **областта на действие** (**scope**) на едно поле е от реда, на който е декларирано, до затварящата фигурна скоба на тялото на класа.

Инициализация по време на деклариране

Когато декларираме едно поле е възможно едновременно с неговата декларация да му дадем първоначална стойност. Начинът, по който става това, е същият както при инициализацията (даването на стойност) на обикновена локална променлива:

```
[<modifiers>] <field_type> <field_name> = <initial_value>;
```

Разбира се, трябва **`<initial_value>`** да бъде от типа на полето или някой съвместим с него тип. Например:

```
class SampleClass
{
   int age = 5;
   long distance = 234; // The literal 234 is of integer type

   string[] names = new string[] { "Pencho", "Marincho" };
   Dog myDog = new Dog();

   // ... Other code ...
}
```

Стойности по подразбиране на полетата

Всеки път, когато създаваме нов обект от даден клас, се заделя област в динамичната памет за всяко поле от класа. След като бъде заделена, тази памет се инициализира автоматично с подразбиращи стойности за конкретния тип поле (занулява се). Полетата, които на се инициализират изрично при декларацията на полето или в някой от конструкторите, се зануляват.

При създаване на обект всички негови полета се инициализират с подразбиращите се стойности за типа им, освен ако изрично не бъдат инициализирани.

В някои езици (като C и C++) новозаделените обекти не се инициализират автоматично с нулеви стойности и това създава условия за допускане на трудни за откриване грешки. Появява се синдромът "ама това вчера работеше" – непредвидимо поведение, при което програмата понякога работи коректно (когато заделената памет съдържа по случайност благоприятни стойности), а понякога не работи (когато заделената памет съдържа неблагоприятни стойности. В C# и въобще в .NET платформата този проблем е решен чрез автоматичното зануляване на полетата.

Стойността по подразбиране за всички типове е 0 или неин еквивалент. За най-често използваните типове подразбиращите се стойности са както следва:

Тип на поле	Стойност по подразбиране
bool	**false**
byte	**0**
char	**'\0'**
decimal	**0.0M**
double	**0.0D**
float	**0.0F**
int	**0**
референция към обект	**null**

За по-изчерпателна информация може да погледнете темата "Примитивни типове и променливи", секция "Типове данни", подсекция "Видове", където има пълен списък с всички примитивни типове данни в C# и подразбиращите се стойности за всеки един от тях.

Например, ако създадем клас **Dog** и за него дефинираме полета име (**name**), възраст (**age**), дължина (**length**) и дали кучето е от мъжки пол (**isMale**), без да ги инициализираме по време на декларацията им, те ще бъдат автоматично занулени при създаването на обект от този клас:

```
public class Dog
{
    string name;
    int age;
    int length;
    bool isMale;

    static void Main()
    {
        Dog dog = new Dog();

        Console.WriteLine("Dog's name is: " + dog.name);
        Console.WriteLine("Dog's age is: " + dog.age);
        Console.WriteLine("Dog's length is: " + dog.length);
        Console.WriteLine("Dog is male: " + dog.isMale);
    }
}
```

Съответно при стартиране на примера като резултат ще получим:

```
Dog's name is:
Dog's age is: 0
Dog's length is: 0
Dog is male: False
```

Автоматична инициализация на локални променливи и полета

Ако дефинираме дадена локална променлива в един метод, без да я инициализираме, и веднага след това се опитаме да я използваме (примерно като отпечатаме стойността ѝ), това ще предизвика грешка при компилация, тъй като локалните променливи не се инициализират с подразбиращи се стойности по време на тяхното деклариране.

За разлика от полетата, локалните променливи, не биват инициализирани с подразбираща се стойност при тяхното деклариране.

Нека разгледаме един пример:

```
static void Main()
{
   int notInitializedLocalVariable;
   Console.WriteLine(notInitializedLocalVariable);
}
```

Ако се опитаме да компилираме горния код, ще получим следното съобщение за грешка:

```
Use of unassigned local variable 'notInitializedLocalVariable'
```

Собствени стойности по подразбиране

Добър стил на програмиране е обаче, когато декларираме полетата на класа си, изрично да ги инициализираме с някаква подразбираща се стойност, дори ако тя е нула. Въпреки, че C# ще занули всяко едно от полетата, ако ги инициализираме изрично, ще направим кода по-ясен и по-лесен за възприемане.

Пример за такова инициализиране може да дадем като модифицираме класът **SampleClass** от предходната секция "Инициализация по време на деклариране":

```
class SampleClass
{
   int age = 0;
   long distance = 0;
   string[] names = null;
   Dog myDog = null;
```

```
  // ... Other code ...
}
```

Модификатори const и readonly

Както споменахме в началото на тази секция, в декларацията на едно поле е позволено да се използват модификаторите **`const`** и **`readonly`**. Те не са модификатори за достъп, а се използват за еднократно инициализиране на полета. Полета, декларирани като **`const`** или **`readonly`** се наричат **константи**. Използват се когато дадена стойност се повтаря на няколко места в програмата. В такива стойността се изнася като константа и се дефинира само веднъж. Пример за константи от .NET Framework са математическите константи **`Math.PI`** и **`Math.E`**, както и константите **`String.Empty`** и **`Int32.MaxValue`**.

Константи, декларирани с const

Полетата, имащи модификатор **`const`** в декларацията си, трябва да бъдат инициализирани при декларацията си и след това стойността им не може да се променя. Те могат да бъдат достъпвани без да има инстанция на класа, тъй като са споделени между всичко обекти на класа. Нещо повече, при компилация на всички места в кода, където се реферират **`const`** полета, те се заместват със стойността им, сякаш тя е зададена директно, а не чрез константа. По тази причина **`const`** полетата се наричат още **compile-time константи**, защото се заместват със стойността им по време на компилация.

Константи, декларирани с readonly

Модификаторът **`readonly`** задава полета, чиято стойността не може да се променя след като веднъж е зададена. Полетата, декларирани с **`readonly`**, позволяват еднократна инициализация или в момента на декларирането им или в конструкторите на класа. По-късно те не могат да се променят. По тази причина **`readonly`** полетата се наричат още **run-time константи** – константи, защото стойността им не може да се променя след като се зададе първоначално и run-time, защото стойността им се извлича по време на работа на програмата, както при всички останали полета в класа.

Нека онагледим казаното с пример:

```
public class ConstReadonlyModifiersTest
{
  public const double PI = 3.1415926535897932385;
  public readonly double size;

  public ConstReadonlyModifiersTest(int size)
```

```
   {
      this.size = size; // Cannot be further modified!
   }

   static void Main()
   {
      Console.WriteLine(PI);
      Console.WriteLine(ConstReadonlyModifiersTest.PI);
      ConstReadonlyModifiersTest t =
         new ConstReadonlyModifiersTest(5);
      Console.WriteLine(t.size);

      // This will cause compile-time error
      Console.WriteLine(ConstReadonlyModifiersTest.size);
   }
}
```

Методи

В главата "Методи" подробно се запознахме с това как да декларираме и използваме метод. В тази секция накратко ще припомним казаното там и ще се фокусираме върху някои допълнителни особености при декларирането и създаването на методи.

Деклариране на методи в даден клас

Декларирането на методи, както знаем, става по следния начин:

```
// Method definition
[<modifiers>] [<return_type>] <method_name>([<parameters_list>])
{
   // ... Method’s body ...
   [<return_statement>];
}
```

Задължителните елементи при декларирането на метода са типът на връщаната стойност **`<return_type>`**, името на метода **`<method_name>`** и отварящата и затварящата кръгли скоби – "`(`" и "`)`".

Списъкът от параметри **`<params_list>`** не е задължителен. Използваме го да подаваме някакви данни на метода, който декларираме, ако той има нужда.

Знаем, че ако типът на връщаната стойност **`<return_type>`** е **`void`**, тогава **`<return_statement>`** може да участва само с оператора **`return`** без аргумент, с цел прекратяване действието на метода. Ако **`<return_type>`** е различен от **`void`**, методът задължително трябва да връща резултат чрез ключовата дума **`return`** с аргумент, който е от тип **`<return_type>`** или съвместим с него.

Работата, която методът трябва да свърши, се намира в тялото му, заградена от фигурни скоби – "**{**" и "**}**".

Макар че разгледахме някои от модификаторите за достъп, позволени да се използват при декларирането на един метод, в секцията "Видимост на полета и методи" ще разгледаме по-подробно тази тема.

Ще разгледаме модификатора **`static`** в секцията "Статични класове (Static classes) и статични членове на класа (static members) на тази глава.

Пример – деклариране на метод

Нека погледнем декларирането на един метод за намиране сбор на две цели числа:

```
int Add(int number1, int number2)
{
   int result = number1 + number2;
   return result;
}
```

Името, с което сме го декларирали, е **`Add`**, а типът на връщаната му стойност е **`int`**. Списъкът му от параметри се състои от два елемента – променливите **`number1`** и **`number2`**. Съответно, връщаме стойността на сбора от двете числа като резултат.

Достъп до нестатичните данни на класа

В главата "Създаване и използване на обекти", разгледахме как чрез оператора точка, можем да достъпим полетата и да извикаме методите на един клас. Нека припомним как можем да достъпваме полета и да извикваме методи на даден клас, които не са статични, т.е. нямат модификатор **`static`**, в декларацията си.

Например, нека имаме клас **`Dog`**, с поле за възраст – **`age`**. За да отпечатаме стойността на това поле, е нужно да създадем обект от клас **`Dog`** и да достъпим полето на този обект чрез точкова нотация:

```
public class Dog
{
   int age = 2;

   public static void Main()
   {
      Dog dog = new Dog();
      Console.WriteLine("Dog's age is: " + dog.age);
   }
}
```

Съответно резултатът ще бъде:

```
Dog's age is: 2
```

Достъп до нестатичните полетата на класа от нестатичен метод

Достъпът до стойността на едно поле може да се осъществява не директно чрез оператора точка (както бе в последния пример **dog.age**), а чрез метод или свойство. Нека в класа **Dog** си създадем метод, който връща стойността на полето **age**:

```
public int GetAge()
{
   return this.age;
}
```

Както виждаме, за да достъпим стойността на полето за възрастта, вътре, от самия клас, използваме ключовата дума **this**. Знаем, че ключовата дума **this** е референция към текущия обект, към който се извиква метода. Следователно, в нашия пример, с "**return this.age**", ние казваме "от текущия обект (**this**) вземи (използването на оператора точка) стойността на полето **age** и го върни като резултат от метода (чрез ключовата дума **return**)". Тогава, вместо в метода **Main()** да достъпваме стойността на полето **age** на обекта **dog**, ние просто ще извикаме метода **GetAge()**:

```
static void Main()
{
   Dog dog = new Dog();
   Console.WriteLine("Dog's age is: " + dog.GetAge());
}
```

Резултатът след тази промяна ще бъде отново същият.

Формално, декларацията за достъп до поле в рамките на класа, е следната:

```
this.<field_name>
```

Нека подчертаем, че този достъп е възможен само от нестатичен код, т.е. метод или блок, който няма модификатор **static**.

Освен за извличане на стойността на едно поле, можем да използваме ключовата дума **this** и за модифициране на полето.

Например, нека декларираме метод **MakeOlder()**, който извикваме всяка година на датата на рождения ден на нашия домашен любимец и който, увеличава възрастта му с една година:

```
public void MakeOlder()
{
   this.age++;
}
```

За да проверим дали това, което написахме работи коректно, в края на метода **`Main()`** добавяме следните два реда:

```
// One year later, on the birthday date...
dog.MakeOlder();
Console.WriteLine("After one year dog's age is: " + dog.age);
```

След изпълнението, резултатът е следният:

```
Dog's age is: 2
After one year dog's age is: 3
```

Извикване нестатичните методи на класа от нестатичен метод

По подобие на полетата, които нямат **`static`** в декларацията си, методите, които също не са статични, могат да бъдат извиквани в тялото на класа чрез ключовата дума **`this`**. Това става, след като към нея, чрез точкова нотация добавим метода, който ни е необходим заедно с аргументите му (ако има параметри):

```
this.<method_name>(…)
```

Например, нека създадем метод **`PrintAge()`**, който отпечатва възрастта на обекта от тип **`Dog`**, като за целта извиква метода **`GetAge()`**:

```
public void PrintAge()
{
   int myAge = this.GetAge();
   Console.WriteLine("My age is: " + myAge);
}
```

На първия ред от примера указваме, че искаме да получим възрастта (стойността на полето **`age`**) на текущия обект, използвайки метода **`GetAge()`** на текущия обект. Това става чрез ключовата дума **`this`**.

Достъпването на нестатичните елементи на класа (полета и методи) се осъществява чрез ключовата дума `this` и оператора за достъп – точка.

Достъп до нестатични данни на класа без използване на this

Когато достъпваме полетата на класа или извикваме нестатичните му методи, е възможно, да го направим без ключовата дума **`this`**. Тогава двата метода, които декларирахме могат да бъдат записани по следния начин:

```
public int GetAge()
{
   return age; // The same like this.age
}

public void MakeOlder()
{
   age++; // The same like this.age++
}
```

Ключовата дума **`this`** се използва, за да укаже **изрично**, че трябва да се осъществи достъп до нестатично поле на даден клас или извикваме негов нестатичен метод. Когато това изрично уточнение не е необходимо, може да бъде пропускана и директно да се достъпва елемента на класа.

⚠	**Когато не е нужно изрично да се укаже, че се осъществява достъп до елемент на класа, ключовата дума `this` може да бъде пропусната.**

Въпреки, че се подразбира, ключовата дума **`this`** често се използва при достъп до полетата на класа, защото прави кода по-лесен за четене и разбиране, като изрично уточнява, че трябва да се направи достъп до член на класа, а не до локална променлива.

Припокриване на полета с локални променливи

От секцията "Деклариране на полета в даден клас" по-горе, знаем, че областта на действие на едно поле е от реда, на който е декларирано полето, до затварящата скоба на тялото на класа. Например:

```
public class OverlappingScopeTest
{
   int myValue = 3;

   void PrintMyValue()
   {
      Console.WriteLine("My value is: " + myValue);
   }

   static void Main()
   {
      OverlappingScopeTest instance = new OverlappingScopeTest();
      instance.PrintMyValue();
   }
}
```

Този код ще изведе в конзолата като резултат:

```
My value is: 3
```

От друга страна, когато имплементираме тялото на един метод, ни се налага да дефинираме локални променливи, които да използваме по време на изпълнение на метода. Както знаем, областта на действие на тези локални променливи започва от реда, на който са декларирани и продължава до затварящата фигурна скоба на тялото на метода. Например, нека добавим този метод в току-що декларирания клас **`OverlappingScopeTest`**:

```
int CalculateNewValue(int newValue)
{
   int result = myValue + newValue;
   return result;
}
```

В този случай, локалната променлива, която използваме, за да изчислим новата стойност, е **`result`**.

Понякога обаче, може името на някоя локална променлива да съвпадне с името на някое поле. Тогава настъпва колизия.

Нека първо погледнем един пример, преди да обясним за какво става въпрос. Нека модифицираме метода **`PrintMyValue()`** по следния начин:

```
void PrintMyValue()
{
   int myValue = 5;
   Console.WriteLine("My value is: " + myValue);
}
```

Ако декларираме така метода, дали той ще се компилира? А ако се компилира, дали ще се изпълни? Ако се изпълни коя стойност ще бъде отпечатана – тази на полето или тази на локалната променлива?

Така деклариран, след като бъде изпълнен методът **`Main()`**, резултатът, който ще бъде отпечатан, ще бъде:

```
My value is: 5
```

Това е така, тъй като C# позволява да се дефинират локални променливи, чиито имена съвпадат с някое поле на класа. Ако това се случи, казваме, че областта на действие на локалната променлива припокрива областта на действие на полето (**scope overlapping**).

Точно затова областта на действие на локалната променлива **myValue** със стойност **5** препокри областта на действие на полето със същото име. Тогава, при отпечатването на стойността, бе използвана стойността на локалната променлива.

Въпреки това, понякога се налага при колизия на имената да бъде използвано полето, а не локалната променлива със същото име. В този случай, за да извлечем стойността на полето, използваме ключовата дума **this**. За целта достъпваме полето чрез оператора точка, приложен към **this**. По този начин еднозначно указваме, че искаме да използваме стойността на полето, не на локалната променлива със същото име.

Нека разгледаме отново нашия пример с извеждането на стойността на полето **myValue**:

```
void PrintMyValue()
{
   int myValue = 5;
   Console.WriteLine("My value is: " + this.myValue);
}
```

Този път, резултатът от извикването на метода е:

```
My value is: 3
```

Видимост на полета и методи

В началото на главата разгледахме общите положения с модификаторите и нивата на достъп на елементите на един клас в C#. По-късно се запознахме подробно с нивата на достъп при декларирането на един клас.

Сега ще разгледаме нивата на видимост на полетата и методите в класа. Тъй като полетата и методите са елементи (членове) на класа и имат едни и същи правила при определяне на нивото им на достъп, ще изложим тези правила едновременно.

За разлика от декларацията на клас, при декларирането на полета и методи на класа, могат да бъдат използвани и четирите нива на достъп – **public**, **protected**, **internal** и **private**. Нивото на видимост **protected** няма да бъде разглеждано в тази глава, тъй като е обвързано с наследяването на класове и е обяснено подробно в главата "Принципи на обектно-ориентираното програмиране".

Преди да продължим, нека припомним, че ако един клас **A**, не е видим (няма достъп) от друг клас **B**, тогава нито един елемент (поле или метод) на класа **A**, не може да бъде достъпен от класа **B**.

Ако два класа не са видими един за друг, то елементите им (полета и методи) не са видими също, независимо с

	какви нива на достъп са декларирани самите те.

В следващите подсекции, към обясненията, ще разглеждаме примери, в които имаме два класа (**`Dog`** и **`Kid`**), които са видими един за друг, т.е. всеки един от класовете може да създава обекти от тип – другия клас и да достъпва елементите му в зависимост от нивото на достъп, с което са декларирани. Ето как изглежда първия клас **`Dog`**:

```
public class Dog
{
   private string name = "Sharo";

   public string Name
   {
      get { return this.name; }
   }

   public void Bark()
   {
      Console.WriteLine("wow-wow");
   }

   public void DoSth()
   {
      this.Bark();
   }
}
```

В освен полета и методи се използва и свойство Name, което просто връща полето name. Ще разгледаме свойствата след малко, така че за момента се фокусирайте върху останалото.

Кодът на класа **`Kid`** има следния вид:

```
public class Kid
{
   public void CallTheDog(Dog dog)
   {
      Console.WriteLine("Come, " + dog.Name);
   }

   public void WagTheDog(Dog dog)
   {
      dog.Bark();
   }
}
```

В момента, всички елементи (полета и методи) на двата класа са декларирани с модификатор за достъп **`public`**, но при обяснението на различните нива на достъп, ще го променяме съответно. Това, което ще ни интересува, е как промяната в нивото на достъп на елементите (полета и методи) на класа **Dog** и ще рефлектира върху достъпа до тези елементи, когато този достъп се извършва от:

- Самото тяло на класа **Dog**.
- Тялото на класа **Kid**, съответно вземайки предвид дали **Kid** е в пространството от имена (или асембли), в което се намира класа **Dog** или не.

Ниво на достъп public

Когато метод или променлива на класа са декларирани с модификатор за достъп **public**, те могат да бъдат достъпвани от други класове, независимо дали другите класове са декларирани в същото пространство от имена, в същото асембли или извън него.

Нека разгледаме двата типа достъп до член на класа, които се срещат в нашите класове **Dog** и **Kid**:

D	Достъп до член на класа осъществен в самата декларация на класа.
R	Достъп до член на класа осъществен, чрез референция към обект, създаден в тялото на друг клас

Когато членовете на двата класа са **public**, се получава следното:

Dog.cs

```
    class Dog
    {
      public string name = "Sharo";

      public string Name
      {
D       get { return this.name; }
      }

      public void Bark()
      {
        Console.WriteLine("wow-wow");
      }

      public void DoSth()
      {
D       this.Bark();
      }
    }
```

Kid.cs

Ⓡ Ⓡ

```
class Kid
{
   public void CallTheDog(Dog dog)
   {
      Console.WriteLine("Come, " + dog.name);
   }

   public void WagTheDog(Dog dog)
   {
      dog.Bark();
   }
}
```

Както виждаме, без проблем осъществяваме, достъп до полето **`name`** и до метода **`Bark()`** в класа **`Dog`** от тялото на самия клас. Независимо дали класът **`Kid`** е в пространството от имена на класа **`Dog`**, можем от тялото му, да достъпим полето **`name`** и съответно да извикаме метода **`Bark()`** чрез оператора точка, приложен към референцията **`dog`** към обект от тип **`Dog`**.

Ниво на достъп internal

Когато член на някой клас бъде деклариран с ниво на достъп **`internal`**, тогава този елемент на класа може да бъде достъпван от всеки клас в същото асембли (т.е. в същия проект във Visual Studio), но не и от класовете извън него (т.е. от друг проект във Visual Studio):

Dog.cs

Ⓓ Ⓓ

```
class Dog
{
   internal string name = "Sharo";

   public string Name
   {
      get { return this.name; }
   }

   internal void Bark()
   {
      Console.WriteLine("wow-wow");
   }

   public void DoSth()
   {
      this.Bark();
   }
}
```

Съответно, за класа **`Kid`**, разглеждаме двата случая:

- Когато е в същото асембли, достъпът до елементите на класа **`Dog`**, ще бъде позволен, независимо дали двата класа са в едно и също пространство от имена или в различни:

Kid.cs

```
class Kid
{
   public void CallTheDog(Dog dog)
   {
      Console.WriteLine("Come, " + dog.name);
   }

   public void WagTheDog(Dog dog)
   {
      dog.Bark();
   }
}
```

- Когато класът **`Kid`** е външен за асемблито, в което е деклариран класът **`Dog`**, тогава достъпът до полето **`name`** и метода **`Bark()`** ще е невъзможен:

Kid.cs

```
class Kid
{
   public void CallTheDog(Dog dog)
   {
      Console.WriteLine("Come, " + dog.name);
   }

   public void WagTheDog(Dog dog)
   {
      dog.Bark();
   }
}
```

Всъщност достъпът до **`internal`** членовете на класа **`Dog`** е невъзможен по две причини: недостатъчна видимост на класа и недостатъчна видимост на членовете му. За да се позволи достъп от друго асембли до класа **`Dog`**, той, е необходимо той да е деклариран като **`public`** и едновременно с това въпросните му членове да са декларирани като **`public`**. Ако или класът или членовете му имат по-ниска видимост, достъпът до тях е невъзможен от други асемблита (други Visual Studio проекти).

Ако се опитаме да компилираме класа **`Kid`**, когато е външен за асемблито, в което се намира класа **`Dog`**, ще получим грешки при компилация.

Ниво на достъп private

Нивото на достъп, което налага най-много ограничения е **`private`**. Елементите на класа, които са декларирани с модификатор за достъп **`private`** (или са декларирани без модификатор за достъп, защото тогава **`private`** се подразбира), не могат да бъдат достъпвани от никой друг клас, освен от класа, в който са декларирани.

Следователно, ако декларираме полето **`name`** и метода **`Bark()`** на класа **`Dog`**, с модификатори **`private`**, няма проблем да ги достъпваме вътрешно от самия клас **`Dog`**, но достъп от други класове не е позволен, дори ако са от същото асембли:

Dog.cs

```
    class Dog
    {
      private string name = "Sharo";

      public string Name
      {
D         get { return this.name; }
      }

      private void Bark()
      {
        Console.WriteLine("wow-wow");
      }

      public void DoSth()
      {
D         this.Bark();
      }
    }
```

Kid.cs

```
    class Kid
    {
      public void CallTheDog(Dog dog)
      {
R         Console.WriteLine("Come, " + dog.name);
      }

      public void WagTheDog(Dog dog)
      {
R         dog.Bark();
      }
    }
```

Трябва да знаем, че когато задаваме модификатор за достъп за дадено поле, той най-често трябва да бъде **private**, тъй като така даваме възможно най-висока защита на достъпа до стойността на полето. Съответно, достъпът и модификацията на тази стойност от други класове (ако са необходими) ще се осъществяват единствено чрез свойства или методи. Повече за тази техника ще научим в секцията "Капсулация" на главата "Принципи на обектно-ориентираното програмиране".

Как се определя нивото на достъп на елементите на класа?

Преди да приключим със секцията за видимостта на елементите на един клас, нека направим един експеримент. Нека в класа **Dog** полето **name** и метода **Bark()** са декларирани с модификатор за достъп **private**. Нека също така, декларираме метод **Main()**, със следното съдържание:

```
public class Dog
{
   private string name = "Sharo";

   // ...

   private void Bark()
   {
      Console.WriteLine("wow-wow");
   }

   // ...

   public static void Main()
   {
      Dog myDog = new Dog();
      Console.WriteLine("My dog's name is " + myDog.name);
      myDog.Bark();
   }
}
```

Въпросът, който стои пред нас е, ще се компилира ли класът **Dog**, при положение, че сме декларирали елементите на класа с модификатор за достъп **private**, а в същото време ги извикваме с точкова нотация, приложена към променливата **myDog**, в метода **Main()**?

Стартираме компилацията и тя минава **успешно**. Съответно, резултатът от изпълнението на метода **Main()**, който декларирахме в класа **Dog** ще бъде следният:

```
My dog’s name is Sharo
Wow-wow
```

Всичко се компилира и работи, тъй като модификаторите за достъп до елементите на класа се прилагат на ниво клас, а не на ниво обекти. Тъй като променливата `myDog` е дефинирана в тялото на класа `Dog` (където е разположен и `Main()` метода на програмата), можем да достъпваме елементите му (полета и методи) чрез точкова нотация, независимо че са декларирани с ниво на достъп `private`. Ако обаче се опитаме да направим същото от тялото на класа `Kid`, това няма да е възможно, тъй като достъпът до `private` полетата от външен клас не е разрешено.

Конструктори

В обектно-ориентираното програмиране, когато създаваме обект от даден клас, е необходимо да извикаме елемент от класа, наречен конструктор.

Какво е конструктор?

Конструктор на даден клас, наричаме псевдометод, който няма тип на връщана стойност, носи името на класа и който се извиква чрез ключовата дума `new`. Задачата на конструктора е да инициализира заделената за обекта памет, в която ще се съхраняват неговите полетата (тези, които не са `static`).

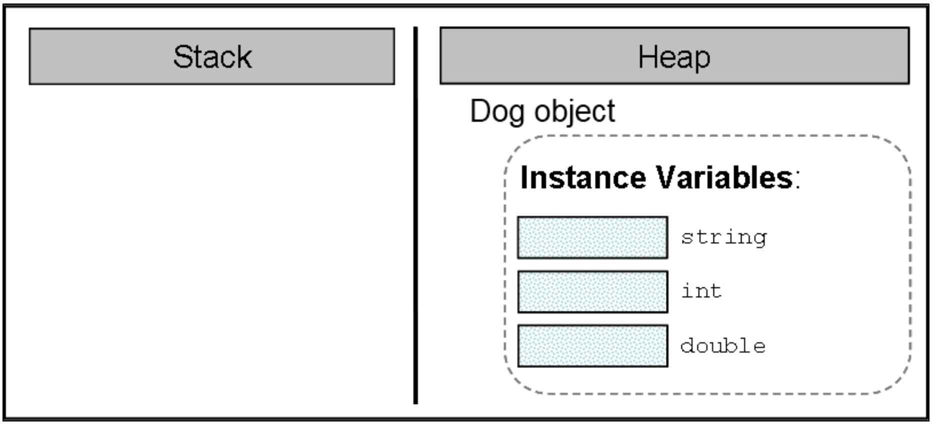
Извикване на конструктор

Единственият начин да извикаме един конструктор в C# е чрез ключовата дума `new`. Тя заделя памет за новия обект (в стека или в хийпа според това дали обектът е стойностен или референтен тип), занулява полетата му, извиква конструктора му (или веригата конструктори, образувана при наследяване) и накрая връща референция към новозаделения обект.

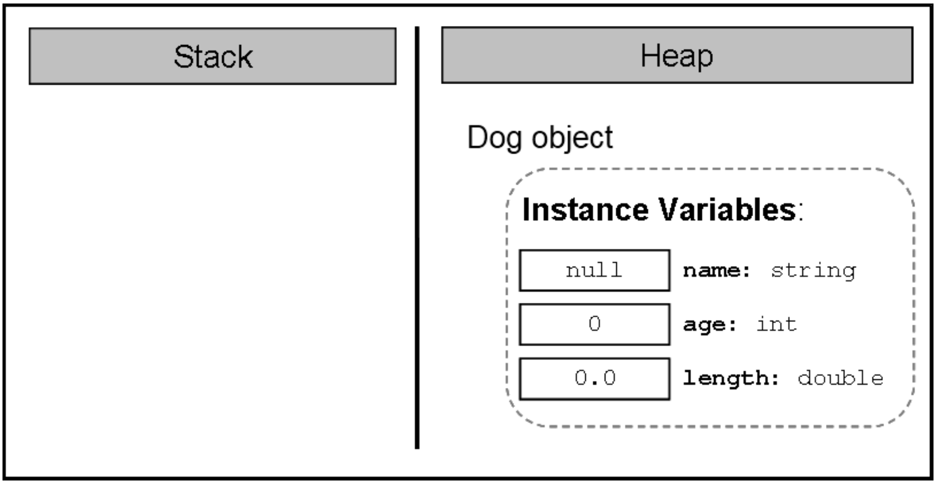
Нека разгледаме един пример, от който ще стане ясно как работи конструкторът. От главата "Създаване и използване на обекти", знаем как се създава обект:

```
Dog myDog = new Dog();
```

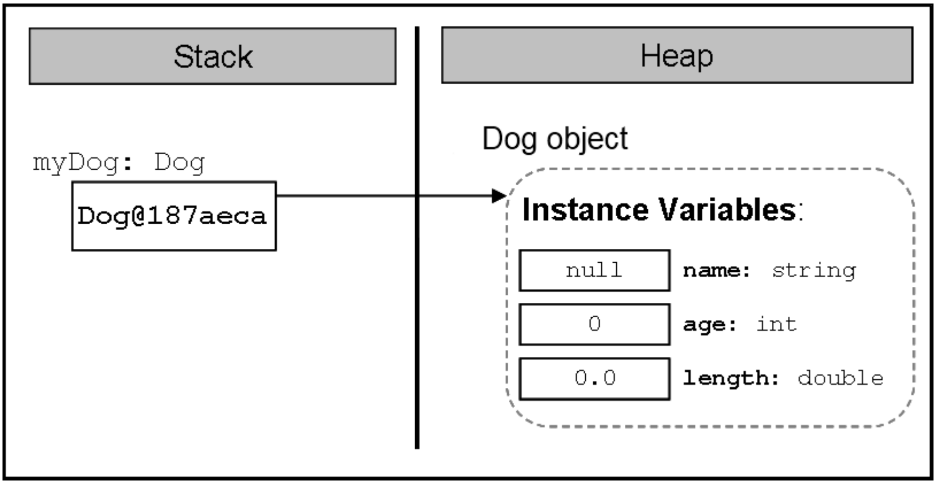
В случая, чрез ключовата дума `new`, извикваме конструктора на класа `Dog`, при което се заделя паметта необходима за новосъздадения обект от тип `Dog`. Когато става дума за класове, те се заделят в динамичната памет (хийпа). Нека проследим как протича този процес стъпка по стъпка. Първо се заделя памет за обекта:


Stack
Heap
Dog object
Instance Variables:
string
int
double

След това се инициализират полетата му (ако има такива) с подразбиращите се стойности за съответните им типове:


Stack
Heap
Dog object
Instance Variables:
null
name: string
0
age: int
0.0
length: double

Ако създаването на новия обект е завършило успешно, конструкторът връща референция към него, която се присвоява на променливата **`myDog`**, от тип класа **`Dog`**:


Stack
Heap
myDog: Dog
Dog@187aeca
Dog object
Instance Variables:
null
name: string
0
age: int
0.0
length: double

Деклариране на конструктор

Ако имаме класа **`Dog`**, ето как би изглеждал неговия най-опростен конструктор:

```
public Dog()
{
}
```

Формално, декларацията на конструктора изглежда по следния начин:

```
[<modifiers>] <class_name>([<parameters_list>])
```

Както вече знаем, конструкторите приличат на методи, но нямат тип на връщана стойност (затова ги нарекохме псевдометоди).

Име на конструктора

В C# задължително името на всеки конструктор съвпада с името на класа, в който го декларираме – **`<class_name>`**. В примера по-горе, името на конструктора е същото, каквото е името на класа – **`Dog`**. Трябва да знаем, че както при методите, името на конструктора винаги е следвано от кръгли скоби – "(" и ")".

В C# **не е позволено**, да се декларира метод, който притежава име, което съвпада с името на класа (следователно и с името на конструкторите). Ако въпреки всичко бъде деклариран метод с името на класа, това ще доведе до грешка при компилация.

```
public class IllegalMethodExample
{
   // Legal constructor
   public IllegalMethodExample ()
```

```
    {
    }

    // Illegal method
    private string IllegalMethodExample()
    {
      return "I am illegal method!";
    }
}
```

При опит за компилация на този клас, компилаторът ще изведе следното съобщение за грешка:

```
SampleClass: member names cannot be the same as their enclosing
type
```

Списък с параметри

По подобие на методите, ако за създаването на обекта са необходими допълнителни данни, конструкторът ги получава чрез списък от параметри – **`<parameters_list>`**. В примерния конструктор на класа **`Dog`** няма нужда от допълнителни данни за създаване на обект от такъв тип и затова няма деклариран списък от параметри. Повече за списъка от параметри ще разгледаме в една от следващите секции – "Деклариране на конструктор с параметри".

Разбира се след декларацията на конструктора, следва неговото тяло, което е като тялото на всеки един метод в C#, но по принцип съдържа предимно инициализационна логика, т.е. задава начални стойности на полетата на класа.

Модификатори

Забелязваме, че в декларацията на конструктора, може да се добавят модификатори – **`<modifiers>`**. За модификаторите, които познаваме и които не са модификатори за достъп, т.е. **`const`** и **`static`**, трябва да знаем, че само **`const`** не е позволен за употреба при декларирането на конструктори. По-късно в тази глава, в секцията "Статични конструктори" ще научим повече подробности за конструктори декларирани с модификатор **`static`**.

Видимост на конструкторите

По подобие на полетата и методите на класа, конструкторите, могат да бъдат декларирани с нива на видимост **`public`**, **`protected`**, **`internal`**, **`protected internal`** и **`private`**. Нивата на достъп **`protected`** и **`protected internal`** ще бъдат обяснени в главата "Принципи на обектно-ориентираното програмиране". Останалите нива на достъп имат същото значение и поведение като при полетата и методите.

Инициализация на полета в конструктора

Както обяснихме по-рано, при създаването на нов обект и извикването на конструктор, се заделя памет за нестатичните полетата на обекта от дадения клас и те се инициализират със стойностите по подразбиране за техния тип (вж. секция "Извикване на конструктор").

Освен това, чрез конструкторите най-често инициализираме полетата на класа, със стойности зададени от нас, а не с подразбиращите се за типа.

Например, в примерите, които разглеждахме до момента, винаги полето **`name`** на обекта от тип **`Dog`**, го инициализирахме по време на неговата декларация:

```
string name = "Sharo";
```

Вместо да правим това по време на декларацията на полето, по-добър стил на програмиране е да му дадем стойност в конструктора:

```
public class Dog
{
   private string name;

   public Dog()
   {
      this.name = "Sharo";
   }

   // ... The rest of the class body ...
}
```

В някои книги се препоръчва, въпреки че инициализираме полетата в конструктора, изрично да присвояваме подразбиращите се за типа им стойности по време на инициализация, с цел да се подобри четимостта на кода, но това е въпрос на личен избор:

```
public class Dog
{
   private string name = null;

   public Dog()
   {
      this.name = "Sharo";
   }

   // ... The rest of the class body ...
}
```

Инициализация на полета в конструктора – представяне в паметта

Нека разгледаме подробно, какво прави конструкторът след като бъде извикан и в тялото му инициализираме полетата на класа. Знаем, че при извикване, той ще задели памет за всяко поле и тази памет ще бъде инициализирана със стойността по подразбиране.

Ако полетата са от примитивен тип, тогава след подразбиращите се стойности, ще бъдат присвоени новите, които ние подаваме.

В случая, когато полетата са от референтен тип, например нашето поле **`name`**, конструкторът ще ги инициализира с **`null`**. След това ще създаде обекта от съответния тип, в случая низа "**`Sharo`**" и накрая ще се присвои референция към новия обект в съответното поле, при нас – полето **`name`**.

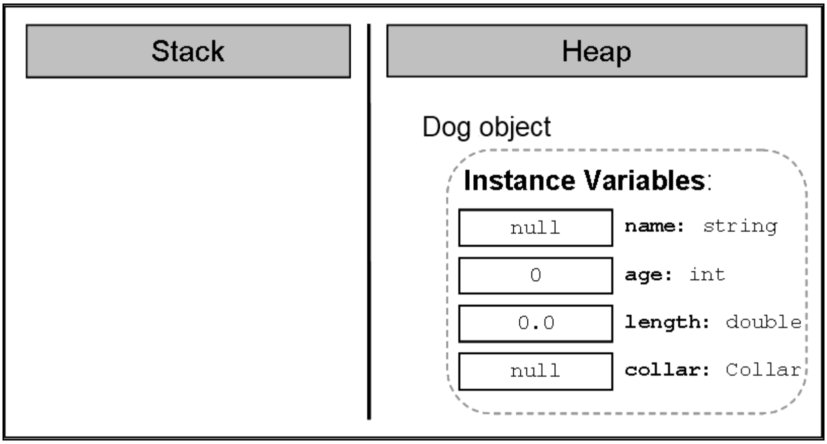
Същото ще се получи, ако имаме и други полета, които не са примитивни типове и ги инициализираме в конструктора. Например, нека имаме клас, който описва каишка – **`Collar`**:

```
public class Collar
{
   private int size;

   public Collar()
   {
   }
}
```

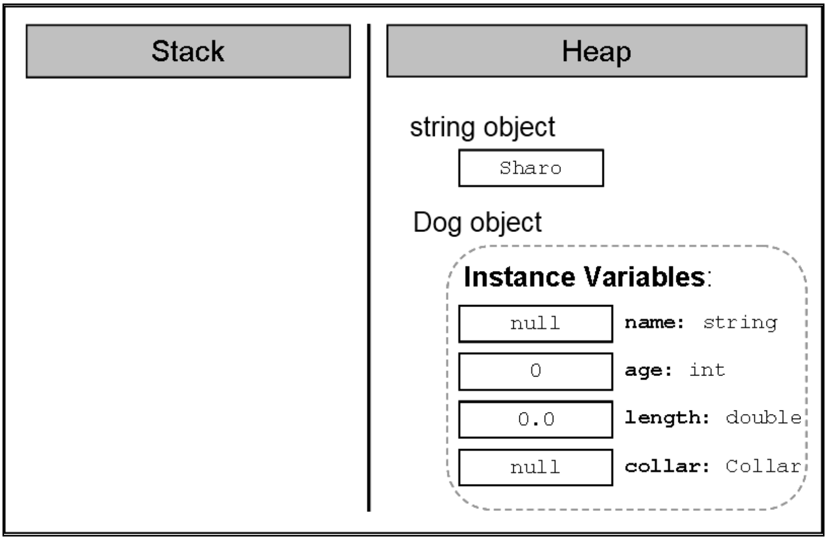
Нека съответно нашият клас **`Dog`**, има поле **`collar`**, което е от тип **`Collar`** и което инициализираме в конструктора на класа:

```
public class Dog
{
   private string name;
   private int age;
   private double length;
   private Collar collar;

   public Dog()
   {
      this.name = "Sharo";
      this.age = 3;
      this.length = 0.5;
      this.collar = new Collar();
   }

   public static void Main()
   {
      Dog myDog = new Dog();
   }
}
```

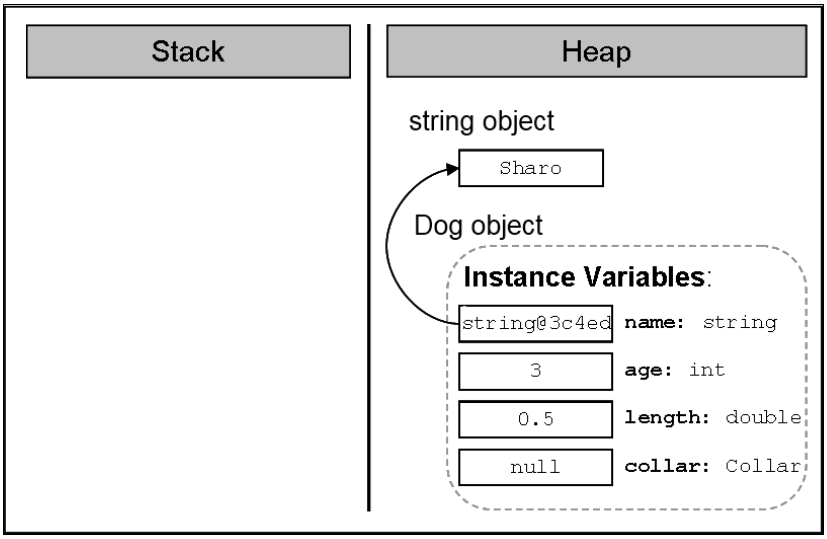
Нека проследим стъпките, през които минава конструкторът, след като бъде извикан в **`Main()`** метода. Както знаем, той ще задели памет в хийпа за всички полета, и ще ги инициализира със съответните им подразбиращи се стойности:


Stack
Heap
Dog object
Instance Variables:
null
name: string
0
age: int
0.0
length: double
null
collar: Collar

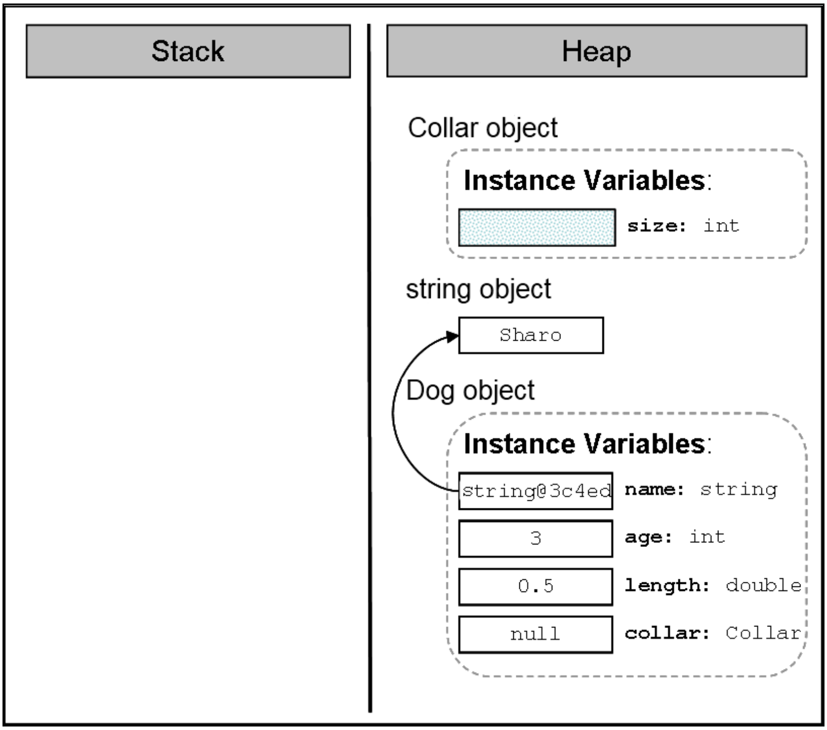
След това, конструкторът ще трябва да се погрижи за създаването на обекта за полето **`name`** (т.е. ще извика конструктора на класа **`string`**, който ще свърши работата по създаването на низа):


Stack
Heap
string object
Sharo
Dog object
Instance Variables:
null
name: string
0
age: int
0.0
length: double
null
collar: Collar

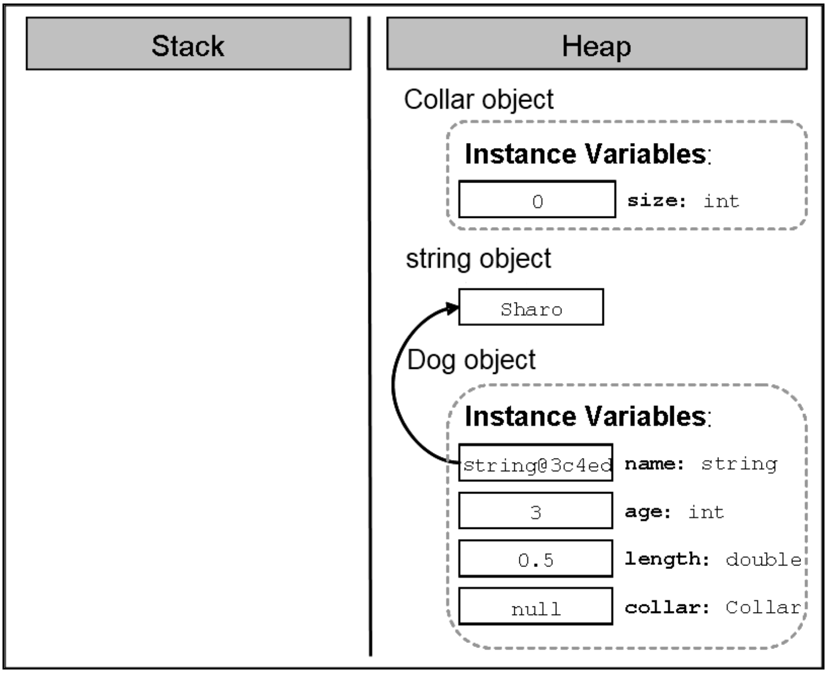
След това нашия конструктор ще запази референция към новия низ в полето **`name`**:


Stack
Heap
string object
Sharo
Dog object
Instance Variables:
string@3c4ed
name: string
3
age: int
0.5
length: double
null
collar: Collar

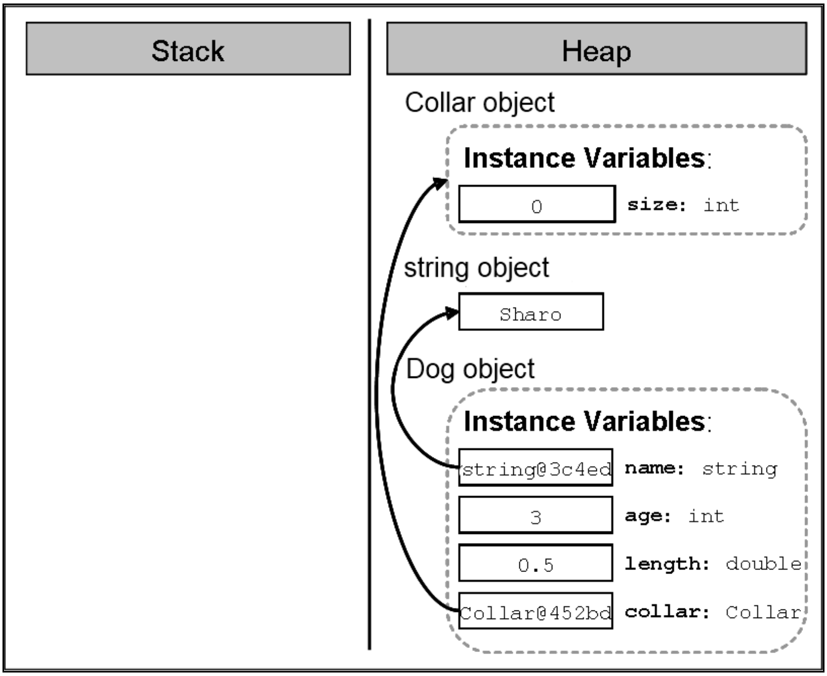
След това идва ред на създаването на обекта от тип **`Collar`**. Нашият конструктор (на класа **`Dog`**), извиква конструктора на класа **`Collar`**, който заделя памет за новия обект:


Stack
Heap
Collar object
Instance Variables:
size: int
string object
Sharo
Dog object
Instance Variables:
string@3c4ed
name: string
3
age: int
0.5
length: double
null
collar: Collar

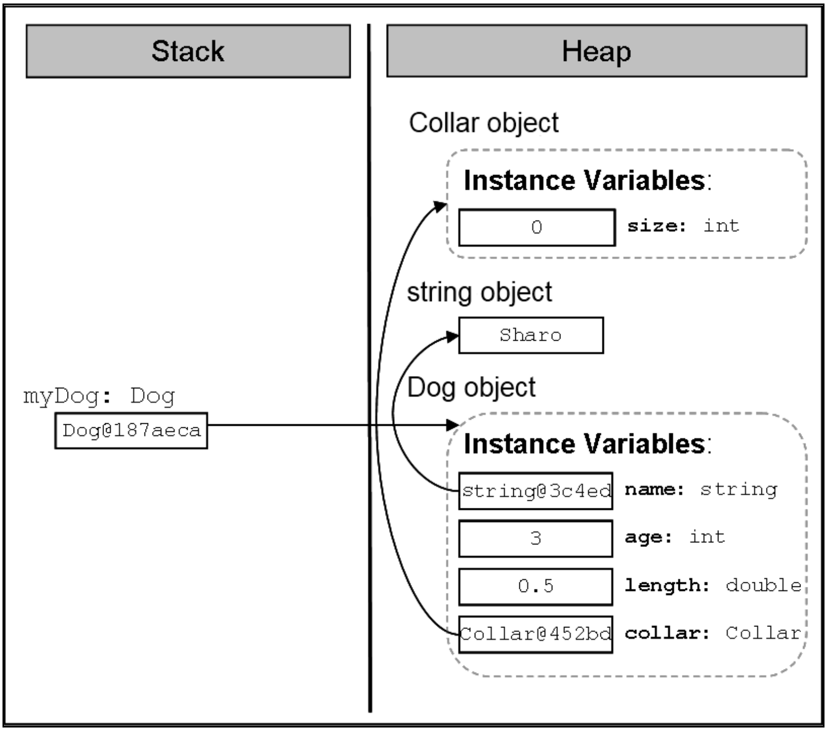
След това я инициализира с подразбиращата се стойност за съответния тип:


Stack
Heap
Collar object
Instance Variables:
0
size: int
string object
Sharo
Dog object
Instance Variables:
string@3c4ed
name: string
3
age: int
0.5
length: double
null
collar: Collar

След това референцията към новосъздадения обект, която конструкторът на класа **`Collar`** връща като резултат от изпълнението си, се записва в полето **`collar`**:


Stack
Heap
Collar object
Instance Variables:
0
size: int
string object
Sharo
Dog object
Instance Variables:
string@3c4ed
name: string
3
age: int
0.5
length: double
Collar@452bd
collar: Collar

Накрая, референцията към новия обект от тип **`Dog`** се присвоява на локалната променлива **`myDog`** в метода **`Main()`**:


Stack
Heap
Collar object
Instance Variables:
0
size: int
string object
Sharo
Dog object
myDog: Dog
Dog@187aeca
Instance Variables:
string@3c4ed
name: string
3
age: int
0.5
length: double
Collar@452bd
collar: Collar

Помним, че локалните променливи винаги се съхраняват в областта от оперативната памет, наречена стек, а обектите – в частта, наречена хийп.

Последователност на инициализиране на полетата на класа

За да няма объркване, нека разясним последователността, в която се инициализират полетата на един клас, независимо от това дали сме им дали стойност по време на декларация и/или сме ги инициализирали в конструктора.

Първо се заделя памет за съответното поле в хийпа и тази памет се инициализира със стойността по подразбиране на типа на полето. Например, нека разгледаме отново нашия клас **`Dog`**:

```
public class Dog
{
   private string name;

   public Dog()
   {
      Console.WriteLine(
         "this.name has value of: \"" + this.name + "\"");
      // ... No other code here ...
   }
   // ... Rest of the class body ...
}
```

При опит да създадем нов обект от тип нашия клас, в конзолата ще бъде отпечатано съответно:

```
this.name has value of: ""
```

Втората стъпка на CLR, след инициализирането на полетата със стойността по подразбиране за съответния тип е, ако е зададена стойност при декларацията на полето, тя да му се присвои.

Така, ако променим реда от класа **`Dog`**, на който декларираме полето **`name`**, то първоначално ще бъде инициализирано със стойност **`null`** и след това ще му бъде присвоена стойността "**`Rex`**".

```
private string name = "Rex";
```

Съответно, при всяко създаване на обект от нашия клас:

```
public static void Main()
{
   Dog dog = new Dog();
}
```

Ще бъде извеждано:

```
this.name has value of: "Rex"
```

Едва след тези две стъпки на инициализация на полетата на класа (инициализиране със стойностите по подразбиране и евентуално стойността зададена от програмиста по време на декларация на полето), се извиква конструкторът на класа. Едва тогава, полетата получават стойностите, с които са им дадени в тялото на конструктора.

Деклариране на конструктор с параметри

В предната секция, видяхме как можем да дадем стойности на полетата, различни от стойностите по подразбиране. Много често, обаче, по време на декларирането на конструктора не знаем какви стойности ще приемат различните полета. За да се справим с този проблем, по подобие на методите с параметри, нужната информация, която трябва за работата на конструктора, му се подава чрез списъка с параметри. Например:

```
public Dog(string dogName, int dogAge, double dogLength)
{
  name = dogName;
  age = dogAge;
  length = dogLength;
  collar = new Collar();
}
```

Съответно, извикването на конструктор с параметри, става по същия начин както извикването на метод с параметри – нужните стойности ги подаваме в списък, чийто елементи са разделени със запетайки:

```
public static void Main()
{
  Dog myDog = new Dog("Bobi", 2, 0.4); // Passing parameters

  Console.WriteLine("My dog " + myDog.name +
    " is " + myDog.age + " year(s) old. " +
      " and it has length: " + myDog.length + " m.");
}
```

Резултатът от изпълнението на този **`Main()`** метод е следния:

```
My dog Bobi is 2 year(s) old. It has length: 0.4 m.
```

В C# нямаме ограничение за броя на конструкторите, които можем да създадем. Единственото условие е те да се различават по сигнатурата си (какво е сигнатура обяснихме в главата "Методи").

Област на действие на параметрите на конструктора

По аналогия на областта на действие на променливите в списъка с параметри на един метод, променливите в списъка с параметри на един конструктор имат област на действие от отварящата скоба на конструктора до затварящата такава, т.е. в цялото тяло на конструктора.

Много често, когато декларираме конструктор с параметри, е възможно да именуваме променливите от списъка му с параметри, със същите имена, като имената на полетата, които ще бъдат инициализирани. Нека за пример вземем отново конструктора, който декларирахме в предходната секция:

```
public Dog(string name, int age, double length)
{
  name = name;
  age = age;
  length = length;
  collar = new Collar();
}
```

Нека компилираме и изпълним съответно **`Main()`** метода, който също използвахме в предходната секция. Ето какъв е резултатът от изпълнението му:

```
My dog  is 0 year(s) old. It has length: 0 m
```

Странен резултат, нали? Всъщност се оказва, че не е толкова странен. Обяснението е следното – областта, в която действат променливите от списъка с параметри на конструктора, припокрива областта на действие на полетата, които имат същите имена в конструктора. По този начин не даваме никаква стойност на полетата, тъй като на практика ние не ги достъпваме. Например, вместо на полето **`age`**, ние присвояваме стойността на променливата **`age`** на самата нея:

```
age = age;
```

Както видяхме в секцията "Припокриване на полета с локални променливи", за да избегнем това разминаване, трябва да достъпим полето, на което искаме да присвоим стойност, но чието име съвпада с името на променлива от списъка с параметри, използвайки ключовата дума **`this`**:

```
public Dog(string name, int age, double length)
{
   this.name = name;
   this.age = age;
   this.length = length;
   this.collar = new Collar();
}
```

Сега, ако изпълним отново **`Main()`** метода:

```
public static void Main()
{
   Dog myDog = new Dog("Bobi", 2, 0.4);

   Console.WriteLine("My dog " + myDog.name +
      " is " + myDog.age + " year(s) old. " +
         " and it has length: " + myDog.length + " m");
}
```

Резултатът ще бъде точно какъвто очакваме да бъде:

```
My dog Bobi is 2 year(s) old. It has length: 0.4 m
```

Конструктор с променлив брой аргументи

Подобно на методите с променлив брой аргументи, които разгледахме в главата "Методи", конструкторите също могат да бъдат декларирани с параметър за променлив брой аргументи. Правилата за декларация и извикване на конструктори с променлив брой аргументи са същите като тези, които описахме за декларацията и извикването при методи:

- Когато декларираме конструктор с променлив брой параметри, трябва да използваме запазената дума **params**, след което поставяме типа на параметрите, следван от квадратни скоби. Накрая, следва името на масива, в който ще се съхраняват подадените при извикване на метода аргументи. Например за целочислени аргументи ползваме **`params int[] numbers`**.
- Позволено е, конструкторът с променлив брой параметри да има и други параметри в списъка си от параметри.
- Параметърът за променлив брой аргументи трябва да е последен в списъка от параметри на конструктора.

Нека разгледаме примерна декларация на конструктор на клас, който описва лекция:

```
public Lecture(string subject, params string[] studentsNames)
{
  // ... Initialization of the instance variables ...
}
```

Първият параметър в декларацията е името на предмета, по който е лекцията, а следващия параметър е за променлив брой аргументи – имената на студентите. Ето как би изглеждало примерното създаване на обект от този клас:

```
Lecture lecture =
  new Lecture("Biology", "Pencho", "Mincho", "Stancho");
```

Съответно, като първи параметър сме подали името на предмета – **`"Biology"`**, а за всички оставащи аргументи – имената на присъстващите студенти.

Варианти на конструкторите (overloading)

Както видяхме, можем да декларираме конструктори с параметри. Това ни дава възможност да декларираме конструктори с различна сигнатура (брой и подредба на параметрите), с цел да предоставим удобство на тези, които ще създават обекти от нашия клас. Създаването на конструктори с различна сигнатура се нарича създаване на **варианти на конструкторите (constructors overloading)**.

Нека вземем за пример класа **`Dog`**. Можем да декларираме различни конструктори:

```
// No parameters
```

```
public Dog()
{
   this.name = "Sharo";
   this.age = 1;
   this.length = 0.3;
   this.collar = new Collar();
}

// One parameter
public Dog(string name)
{
   this.name = name;
   this.age = 1;
   this.length = 0.3;
   this.collar = new Collar();
}

// Two parameters
public Dog(string name, int age)
{
   this.name = name;
   this.age = age;
   this.length = 0.3;
   this.collar = new Collar();
}

// Three parameters
public Dog(string name, int age, double length)
{
   this.name = name;
   this.age = age;
   this.length = length;
   this.collar = new Collar();
}

// Four parameters
public Dog(string name, int age, double length, Collar collar)
{
   this.name = name;
   this.age = age;
   this.length = length;
   this.collar = collar;
}
```

Преизползване на конструкторите

В последния пример, който дадохме, видяхме, че в зависимост от нуждите за създаване на обекти от нашия клас може да декларираме различни варианти на конструктори. Лесно се забелязва, че голяма част от кода на тези конструктори се повтаря. Това ни кара да се замислим, дали няма начин един конструктор, който вече извършва дадена инициализация, да бъде преизползван от другите, които трябва да изпълнят същата инициализация. От друга страна, в началото на главата споменахме, че един конструктор не може да бъде извикан по начина, по който се извикват методите, а само чрез ключовата дума **`new`**. Би трябвало да има някакъв начин, иначе много код ще се повтаря излишно.

В C#, съществува механизъм, чрез който един конструктор може да извиква друг конструктор деклариран в същия клас. Това става отново с ключовата дума this, но използвана в друга синтактична конструкция при декларацията на конструкторите:

```
[<modifiers>] <class_name>([<parameters_list_1>])
   : this([<parameters_list_2>])
```

Към познатата ни форма за деклариране на конструктор (първия ред от декларацията показана по-горе), можем да добавим двоеточие, следвано от ключовата дума **`this`**, следвана от скоби. Ако конструкторът, който искаме да извикаме е с параметри, в скобите трябва да добавим списък от параметри **`parameters_list_2`**, които да му подадем.

Ето как би изглеждал кодът от предходната секция, в който вместо да повтаряме инициализацията на всяко едно от полетата, извикваме конструктори, декларирани в същия клас:

```
// No parameters
public Dog()
   : this("Sharo") // Constructor call
{
   // More code could be added here
}

// One parameter
public Dog(string name)
   : this(name, 1) // Constructor call
{
}

// Two parameters
public Dog(string name, int age)
   : this(name, age, 0.3) // Constructor call
{
}

// Three parameters
public Dog(string name, int age, double length)
   : this(name, age, length, new Collar()) // Constructor call
{
}
```

```
// Four parameters
public Dog(string name, int age, double length, Collar collar)
{
  this.name = name;
  this.age = age;
  this.length = length;
  this.collar = collar;
}
```

Както е указано чрез коментар в първия конструктор от примера по-горе, ако е необходимо, в допълнение към извикването на някой от другите конструктори с определени параметри всеки конструктор може в тялото си да добави код, който прави допълнителни инициализации или други действия.

Конструктор по подразбиране

Нека разгледаме следния въпрос – какво става, ако не декларираме конструктор в нашия клас? Как ще създадем обекти от този тип?

Тъй като често се случва даден клас да няма нито един конструктор, този въпрос е решен в езика C#. Когато не декларираме нито един конструктор, компилаторът ще създаде един за нас и той ще се използва при създаването на обекти от типа на нашия клас. Този конструктор се нарича **конструктор по подразбиране** (**default implicit constructor**), който няма да има параметри и ще бъде празен (т.е. няма да прави нищо в допълнение към подразбиращото се зануляване на полетата на обекта).

Когато не дефинираме нито един конструктор в даден клас, компилаторът ще създаде един, наречен конструктор по подразбиране.

Например, нека декларираме класа **`Collar`**, без да декларираме никакъв конструктор в него:

```
public class Collar
{
  private int size;

  public int Size
  {
    get { return size; }
  }
}
```

Въпреки, че нямаме изрично деклариран конструктор без параметри, ще можем да създадем обекти от този клас по следния начин:

```
Collar collar = new Collar();
```

Конструкторът по подразбиране изглежда по следния начин:

```
<access_level> <class_name>() { }
```

Трябва да знаем, че конструкторът по подразбиране винаги носи името на класа **`<class_name>`** и винаги списъкът му с параметри е празен и неговото тяло е празно. Той просто се "подпъхва" от компилатора, ако в класа няма нито един конструктор. Подразбиращият се конструктор обикновено е **`public`** (с изключение на някои много специфични ситуации, при които е **`protected`**).

Конструкторът по подразбиране е винаги без параметри.

За да се уверим, че конструкторът по подразбиране винаги е без параметри, нека направим опит да извикаме подразбиращия се конструктор, като му подадем параметри:

```
Collar collar = new Collar(5);
```

Компилаторът ще изведе следното съобщение за грешка:

```
'Collar' does not contain a constructor that takes 1 arguments
```

Работа на конструктора по подразбиране

Както се досещаме, единственото, което конструкторът по подразбиране ще направи при създаването на обекти от нашия клас, е да занули полетата на класа. Например, ако в класа **`Collar`** не сме декларирали нито един конструктор и създадем обект от него и се опитаме да отпечатаме стойността в полето **`size`**:

```
public static void Main()
{
   Collar collar = new Collar();
   Console.WriteLine("Collar's size is: " + collar.Size);
}
```

Резултатът ще бъде:

```
Collar's size is: 0
```

Виждаме, че стойността, която е запазена в полето **`size`** на обекта **`collar`**, е точно стойността по подразбиране за целочисления тип **`int`**.

Кога няма да се създаде конструктор по подразбиране?

Трябва да знаем, че ако декларираме поне един конструктор в даден клас, тогава компилаторът няма да създаде конструктор по подразбиране.

За да проверим това, нека разгледаме следния пример:

```
public Collar(int size)
   : this()
{
   this.size = size;
}
```

Нека това е единственият конструктор на класа **`Collar`**. В него се опитваме да извикаме конструктор без параметри, надявайки се, че компилаторът ще е създал конструктор по подразбиране за нас (който знаем, че е без параметри). След като се опитаме да компилираме, ще разберем, че това, което се опитваме да направим, е невъзможно:

```
'Collar' does not contain a constructor that takes 0 arguments
```

Ако сме декларирали дори един единствен конструктор в даден клас, компилаторът няма да създаде конструктор по подразбиране за нас.

⚠	**Ако декларираме поне един конструктор в даден клас, компилаторът няма да създаде конструктор по подразбиране за нас.**

Разлика между конструктор по подразбиране и конструктор без параметри

Преди да приключим със секцията за конструкторите, нека поясним нещо много важно:

⚠	**Въпреки че конструкторът по подразбиране и този, без параметри, си приличат по сигнатура, те са напълно различни.**

Разликата се състои в това, че конструкторът по подразбиране (default implicit constructor) се създава от компилатора, ако не декларираме нито един конструктор в нашия клас, а **конструкторът без параметри (default constructor)** го декларираме ние.

Освен това, както обяснихме по-рано, конструкторът по подразбиране винаги ще има ниво на достъп **`protected`** или **`public`**, в зависимост от модификатора на достъп на класа, докато нивото на достъп на конструктора без параметри изцяло зависи от нас – ние го определяме.

Свойства (Properties)

В света на обектно-ориентираното програмиране съществува елемент на класовете, наречен **свойство (property)**, който е нещо средно между поле и метод и служи за по-добра защита на състоянието в класа. В някои езици за обектно-ориентирано програмиране, като C#, Delphi / Free Pascal, Visual Basic, JavaScript, D, Python и др., свойствата са част от езика, т.е. за тях съществува специален механизъм, чрез който се декларират и използват. Други езици, като например Java, не подържат концепцията за свойства и за целта програмистите, трябва да декларират двойка методи (за четене и модификация на свойството), за да се предостави тази функционалност.

Свойствата в C# – представяне чрез пример

Използването на свойства е доказано добра практика и важна част от концепциите на обектно-ориентираното програмиране. Създаването на свойство в програмирането става чрез деклариране на два метода – един за достъп (четене) и един за модификация (записване) на стойността на съответното свойство.

Нека разгледаме един пример. Да си представим, че имаме отново клас **`Dog`**, който описва куче. Характерно свойство за едно куче е, например, цвета му (color). Достъпът до свойството "цвят" на едно куче и съответната му модификация може да осъществим по следния начин:

```
// Getting (reading) a property
string colorName = dogInstance.Color;

// Setting (modifying) a property
dogInstance.Color = "black";
```

Свойства – капсулация на достъпа до полетата

Основната цел на свойствата е да осигуряват капсулация на състоянието на класа, в който са декларирани, т.е. да го защитят от попадане в невалидни състояния.

Капсулация (encapsulation) наричаме скриването на физическото представяне на данните в един клас, така че, ако в последствие променим това представяне, това да не рефлектира върху останалите класове, които използват този клас.

Чрез синтаксиса на C#, това се реализира като декларираме полета (физическото представяне на данните) с възможно най-ограничено ниво на видимост (най-често с модификатор **`private`**) и декларираме достъпът до тези полета (четене и модифициране) да може да се осъществява единствено чрез специални методи за достъп (accessor methods).

Капсулация – пример

За да онагледим какво представлява капсулацията, която предоставят свойствата на един клас, както и какво представляват самите свойства, ще разгледаме един пример.

Нека имаме клас, който представя точка от двумерното пространство със свойства координатите (x, y). Ето как би изглеждал той, ако декларираме всяка една от координатите, като поле:

Point.cs

```
using System;

class Point
{
  private double x;
  private double y;

  public Point(int x, int y)
  {
    this.x = x;
    this.y = y;
  }

  public double X
  {
    get { return x; }
    set { x = value; }
  }

  public double Y
  {
    get { return y; }
    set { y = value; }
  }
}
```

Полетата на обектите от нашия клас (т.е. координатите на точките) са декларирани като private и не могат да бъдат достъпвани чрез точкова нотация. Ако създадем обект от клас **`Point`**, можем да модифицираме и четем свойствата (координатите) на точката, единствено чрез свойствата за достъп до тях:

PointTest.cs

```
using System;

class PointTest
{
   static void Main()
   {
      Point myPoint = new Point(2, 3);

      double myPointXCoordinate = myPoint.X; // Access a property
      double myPointYCoordinate = myPoint.Y; // Access a property

      Console.WriteLine("The X coordinate is: " +
         myPointXCoordinate);
      Console.WriteLine("The Y coordinate is: " +
         myPointYCoordinate);
   }
}
```

Резултатът от изпълнението на този **`Main()`** метод ще бъде следният:

```
The X coordinate is: 2
The Y coordinate is: 3
```

Ако обаче решим, да променим вътрешното представяне на свойствата на точката, например вместо две полета, ги декларираме като едномерен масив с два елемента, можем да го направим, без това да повлияе по някакъв начин на останалите класове от нашия проект:

`Point.cs`

```
using System;

class Point
{
   private double[] coordinates;

   public Point(int xCoord, int yCoord)
   {
      this.coordinates = new double[2];

      // Initializing the x coordinate
      coordinates[0] = xCoord;

      // Initializing the y coordinate
      coordinates[1] = yCoord;
   }

   public double XCoord
   {
      get { return coordinates[0]; }
      set { coordinates[0] = value; }
   }

   public double YCoord
```

```
    {
      get { return coordinates[1]; }
      set { coordinates[1] = value; }
    }
}
```

Резултатът от изпълнението на **`Main()`** метода няма да се промени и резултатът ще бъде същият, без да променяме дори символ в кода на класа **`PointTest`**.

Демонстрираното е добър пример за добра капсулация на данните на един обект предоставена от механизма на свойствата. Чрез тях скриваме вътрешното представяне на информацията, като декларираме свойства / методи за достъп до него и ако в последствие настъпи промяна в представянето, това няма да се отрази на другите класове, които използват нашия клас, тъй като те ползват само свойствата му и не знаят как е представена информацията "зад кулисите".

Разбира се, разгледаният пример демонстрира само една от ползите да се опаковат (обвиват) полетата на класа в свойства. Свойствата позволяват още контрол над данните в класа и могат да проверяват дали присвояваните стойности свойства са коректни по някакви критерии. Например ако имаме свойство максимална скорост на клас **`Car`**, може чрез свойства да наложим изискването стойността й да е в диапазона между 1 и 300 км/ч.

Още за физическото представяне на свойствата в класа

Както видяхме по-горе, свойствата могат да имат различно представяне в един клас на физическо ниво. В нашия пример, информацията за свойствата на класа **`Point`** първоначално беше съхранена в две полета, а след това в едно поле-масив.

Ако обаче решим, вместо да пазим информацията за свойствата на точката в полета, можем да я запазим във файл или в база данни и всеки път, когато се наложи да достъпваме съответното свойство, можем да четем / пишем от файла или базата вместо както в предходните примери да използваме полета на класа. Тъй като свойствата се достъпват чрез специални методи (наречени методи за достъп и модификация или accessor methods), които ще разгледаме след малко, за класовете, които ще използват нашия клас, това как се съхранява информацията няма да има значение (заради добрата капсулация!).

В най-честия случай обаче, информацията за свойствата на класа се пази в поле на класа, което има възможно най-стриктното ниво на видимост **`private`**.

Няма значение по какъв начин физически ще бъде пазена информацията за свойствата в един C# клас, но обикновено това става чрез поле на класа с максимално ограничено ниво на достъп (`private`).

Представяне на свойство без декларация на поле

Нека разгледаме един пример, в който свойството не се пази нито в поле, нито някъде другаде, а се преизчислява при опит за достъп до него.

Нека имаме клас **`Rectangle`**, който представя геометричната фигура правоъгълник. Съответно този клас има две полета – за ширина **`width`** и дължина **`height`**. Нека нашия клас има и още едно свойство – лице (**`area`**). Тъй като винаги чрез дължината и ширината на правоъгълника можем да намерим стойността на свойството "лице", не е нужно да имаме отделно поле в класа, за да пазим тази стойност. По тази причина, можем да си декларираме просто един метод за получаване на лицето, в който пресмятаме формулата за лице на правоъгълник:

Rectangle.cs

```
using System;

class Rectangle
{
   private float height;
   private float width;

   public Rectangle(float height, float width)
   {
      this.height = height;
      this.width = width;
   }

   // Obtaining the value of the property area
   public float Area
   {
      get { return this.height * this.width; }
   }
}
```

Както ще видим след малко, не е задължително едно свойство да има едновременно методи за модификация и за четене на стойността. Затова е позволено да декларираме само метод за четене на свойството **`Area`** на правоъгълника. Няма смисъл от метод, който модифицира стойността на лицето на един правоъгълник, тъй като то е винаги едно и също при определена дължина на страните.

Деклариране на свойства в C#

За да декларираме едно свойство в C#, трябва да декларираме методи за достъп (за четене и промяна) на съответното свойство и да решим по какъв начин ще съхраняваме информацията за това свойство в класа.

Преди да декларираме методите обаче, трябва да декларираме самото свойството в класа. Формално декларацията на свойствата изглежда по следния начин:

```
[<modifiers>] <property_type> <property_name>
```

С **`<modifiers>`** сме означили, както модификаторите за достъп, така и други модификатори (например **`static`**, който ще разгледаме в следващата секция на главата). Те не са задължителна част от декларацията на едно поле.

Типа на свойството **`<property_type>`** задава типа на стойностите на свойството. Може да бъде както примитивен тип (например **`int`**), така и референтен (например масив).

Съответно, **`<property_name>`** е името на свойството. То трябва да започва с главна буква и да удовлетворява правилото **`PascalCase`**, т.е. всяка нова дума, която се долепя в задната част на името на свойството, започва с главна буква. Ето няколко примера за правилно именувани свойства:

```
// MyValue property
public int MyValue { get; set; }

// Color property
public string Color { get; set; }

// X-coordinate property
public double X { get; set; }
```

Тяло на свойство

Подобно на класа и методите, свойствата в C# имат тяло, където се декларират методите за достъп до свойството (accessors).

```
[<modifiers>] <property_type> <property_name>
{
  // ... Property's accessors methods go here
}
```

Тялото на свойството започва с отваряща фигурна скоба "**{**" и завършва със затваряща – "**}**". Свойствата винаги трябва да имат тяло.

Метод за четене на стойността на свойство (getter)

Както обяснихме, декларацията на **метод за четене на стойността на едно свойство** (в литературата наричан още **getter**) се прави в тялото на свойството, като за целта трябва да се спазва следния синтаксис:

```
get { <accessor_body> }
```

Съдържанието на блока ограден от фигурните скоби (`<accessor_body>`) е подобно на съдържанието на произволен метод. В него се декларират действията, които трябва да се извършат за връщане на резултата от метода.

Методът за четене на стойността на едно свойство трябва да завършва с `return` или `throw` операция. Типът на стойността, която се връща като резултат от този метод, трябва да е същият както типа `<property_type>` описан в декларацията на свойството.

Въпреки, че по-рано в тази секция срещнахме доста примери на декларирани свойства с метод за четене на стойността им, нека разгледаме още един пример за свойството "възраст" (`Age`), което е от тип `int` и е декларирано чрез поле в същия клас:

```
private int age;              // Field declaration

public string Age             // Property declaration
{
  get { return this.age; }   // Getter declaration
}
```

Извикване на метод за четене на стойността на свойство

Ако допуснем, че свойството `Age` от последния пример е декларирано в клас от тип `Dog`, извикването на метода за четене на стойността на свойството, става чрез точкова нотация, приложена към променлива от типа, в чийто клас е декларирано свойството:

```
Dog dogInstance = new Dog();
// ...
int dogAge = dogInstance.Age;          // Getter invocation
Console.WriteLine(dogInstance.Age);    // Getter invocation
```

Последните два реда от примера показват, че достъпвайки чрез точкова нотация името на свойството, автоматично се извиква неговият getter метод (методът за четене на стойността му).

Метод за промяна на стойността на свойство (setter)

По подобие на метода за четене на стойността на едно свойство, може да се декларира и **метод за промяна (модификация) на стойността на**

едно свойство (в литературата наричан още **setter**). Той се декларира в тялото на свойството с тип на връщана стойност **`void`** и в него подадената при присвояването стойност е достъпна през неявен параметър **`value`**.

Декларацията се прави в тялото на свойството, като за целта трябва да се спазва следният синтаксис:

```
set { <accessor_body> }
```

Съдържанието на блока ограден от фигурните скоби (**`<accessor_body>`**) е подобно на съдържанието, на произволен метод. В него се декларират действията, които трябва да се извършат за промяна на стойността на свойството. Този метод използва неявен параметър, наречен **`value`**, който е предоставен от C# по подразбиране и който съдържа новата стойност на свойството. Той е от същия тип, от който е свойството.

Нека допълним примера за свойството "възраст" (**`Age`**) в класа **`Dog`**, за да онагледим казаното дотук:

```
private int age;              // Field declaration

public string Age             // Property declaration
{
  get{ return this.age; }
  set{ this.age = value; }    // Setter declaration
}
```

Извикване на метод за промяна на стойността на свойство

Извикването на метода за модификация на стойността на свойството става чрез точкова нотация, приложена към променлива от типа, в чийто клас е декларирано свойството:

```
Dog dogInstance = new Dog();
// ...
dogInstance.Age = 3;          // Setter invocation
```

На последния ред при присвояването на стойността 3 се извиква setter методът на свойството **`Age`**, с което тази стойност се записва в параметъра **`value`** и се подава на setter метода на свойството **`Age`**. Съответно в нашия пример, стойността на променливата **`value`** се присвоява на полето **`age`** от класа **`Dog`**, но в общия случай може да се обработи по по-сложен начин.

Проверка на входните данни на метода за промяна на стойността на свойство

В процеса на програмиране е добра практика данните, които се подават на setter метода за модификация на свойство да бъдат проверявани дали са валидни и в случай че не са да се вземат необходимите "мерки". Най-често при некоректни данни се предизвиква изключение.

Да вземем отново примера с възрастта на кучето. Както знаем, тя трябва да бъде положително число. За да предотвратим възможността някой да присвои на свойството **`Age`** стойност, която е отрицателно число или нула, добавяме следната проверка в началото на setter метода:

```
public int Age
{
   get { return this.age;   }
   set
   {
      // Take precaution: пerform check for correctness
      if (value <= 0)
      {
         throw new ArgumentException(
            "Invalid argument: Age should be a negative number.");
      }
      // Assign the new correct value
      this.age = value;
   }
}
```

В случай, че някой се опита да присвои стойност на свойството **`Age`**, която е отрицателно число или 0, ще бъде хвърлено изключение от тип **`ArgumentException`** с подробна информация какъв е проблемът.

За да се предпази от невалидни данни един клас трябва да проверява подадените му стойности в setter методите на всички свойства и във всички конструктори, както и във всички методи, които могат да променят някое поле на класа. Практиката класовете да се предпазват от невалидни данни и невалидни вътрешни състояния се използва широко в програмирането и е част от концепцията "Защитно програмиране", която ще разгледаме в главата "Качествен програмен код".

Видове свойства

В зависимост от особеностите им, можем да класифицираме свойствата по следния начин:

1. **Само за четене (read-only)**, т.е. тези свойства имат само **`get`** метод, както в примера с лицето на правоъгълник.
2. **Само за модифициране (write-only)**, т.е. тези свойства имат само **`set`** метод, но не и метод за четене на стойността на свойството.
3. И най-честият случай е **read-write**, когато свойството има **методи както за четене, така и за промяна на стойността**.

Алтернативен похват за работа със свойства

Преди да приключим секцията ще отбележим още нещо за свойствата в един клас, а именно – как можем да декларираме свойства в C#, без да използваме стандартния синтаксис, разгледан до момента.

В езици за програмиране като Java, в които няма концепция (и съответно синтактични средства) за работа със свойства, свойствата се декларират чрез двойка методи, отново наречени getter и setter, по подобие на тези, които разгледахме по-горе.

Тези методи трябва да отговарят на следните изисквания:

1. **Методът за четене на стойността на свойство** е метод без параметри, който трябва да връща резултат от тип, идентичен с типа на свойството и името му да е образувано от името на свойството с представка **`Get`**:

```
[<modifiers>] <property_type> Get<property_name>
```

2. Методът за **модификация на стойността на свойство** трябва да има тип на връщаната стойност **`void`**, името му да е образувано от името на свойството с представка **`Set`** и типа на единствения аргумент на метода да бъде идентичен с този на свойството:

```
[<modifiers>] void Set<property_name>(<property_type> par_name)
```

Ако представим свойството **`Age`** на класа **`Dog`** в примера, който използвахме в предходните секции чрез двойка методи, то декларацията на свойството би изглеждала по следния начин:

```
private int age;                // Field declaration

public int GetAge()             // Getter declaration
{
  return this.age;
}

public void SetAge(int age)     // Setter declaration
{
  this.age = age;
}
```

Съответно, четенето и модификацията на свойството **`Age`**, ще се извършва чрез извикване на декларираните методи:

```
Dog dogInstance = new Dog();

// ...
// Getter invocations
int dogAge = dogInstance.GetAge();
Console.WriteLine(dogInstance.GetAge());

// Setter invocation
dogInstance.SetAge(3);
```

Въпреки че представихме тази алтернатива за декларация на свойства, единствената ни цел бе да бъдем изчерпателни и да направим съпоставка с други езици като Java. Лесно се забелязва, че този начин за декларация на свойствата е по-трудно четим и по-неестествен в сравнение с първия, който изложихме. Затова е препоръчително да се използват вградените средства на езика C# за декларация и използване на свойства.

⚠ **При работа със свойства е препоръчително да се използва стандартният механизъм, който C# предлага, а не алтернативният, който се използва в някои други езици.**

Статични класове (static classes) и статични членове на класа (static members)

Когато един елемент на класа е деклариран с модификатор `static`, го наричаме статичен. В C# като статични могат да бъдат декларирани полетата, методите, свойствата, конструкторите и класовете.

По-долу първо ще разгледаме **статичните елементи** на класа, или с други думи полетата, методите, свойствата и конструкторите му и едва тогава ще се запознаем и с концепцията за статичен клас.

За какво се използват статичните елементи?

Преди да разберем принципа, на който работят статичните елементи на класа, нека се запознаем с причините, поради които се налага използването им.

Метод за сбор на две числа

Нека си представим, че имаме клас, в който един метод винаги работи по един и същ начин. Например, нека неговата задача е да получи две числа чрез списъка му от параметри и да върне като резултат сбора им. При такъв сценарий няма да има никакво значение кой обект от този клас ще изпълни този метод, тъй като той винаги ще се държи по един и същ начин – ще събира две числа, независими от извикващия обект. Реално **поведението на метода не зависи от състоянието на обекта** (стойностите в полетата на обекта). Тогава защо е нужно да създаваме обект, за да изпълним този метод, при положение че методът не зависи от никой от обектите от този клас? Защо просто не накараме класа да изпълни този метод?

Брояч на инстанциите от даден клас

Нека разгледаме и друг сценарий. Да кажем, че искаме да пазим в програмата ни текущия брой на обектите, които са били създадени от даден клас. Как ще съхраним тази променлива, която ще пази броя на създадените обекти?

Както знаем, няма да е възможно да я пазим като поле на класа, тъй като при всяко създаване на обект, ще се създава ново копие на това поле за всеки обект, и то ще бъде инициализирано със стойността по подразбиране. Всеки обект ще пази свое поле за индикация на броя на обектите и обектите няма да могат да споделят информацията по между си. Изглежда броячът не трябва да е поле в класа, а някак си да бъде извън него. В следващите подсекции ще разберем как да се справим и с този проблем.

Какво е статичен член?

Формално погледнато, **статичен член (static member)** на класа наричаме всяко поле, свойство, метод или друг член, който има модификатор `static` в декларацията си[1]. Това означава, че полета, методи и свойства маркирани като статични, принадлежат на самия клас, а не на някой конкретен обект от дадения клас.

Следователно, когато маркираме поле, метод или свойство като статични, можем да ги използваме, без да създаваме нито един обект от дадения клас. Единственото, от което се нуждаем е да имаме достъп (видимост) до класа, за да можем да извикваме статичните му методи, или да достъпваме статичните му полета и свойства.

Статичните елементи на класа могат да се използват без да се създава обект от дадения клас.

От друга страна, ако имаме създадени обекти от дадения клас, тогава статичните полета и свойства ще бъдат общи (споделени) за тях и ще има само едно копие на статичното поле или свойство, което се споделя от всички обекти от дадения клас. По тази причина в езика VB.NET вместо ключовата дума `static` със същото значение се ползва ключовата дума `Shared`.

---

[1] Както споменахме по-рано, конструкторите също могат да бъдат декларирани като статични, но тъй като концепцията за статичен конструктор е по-особена, ще ги разгледаме отделно.

Статични полета

Когато създаваме обекти от даден клас, всеки един от тях има различни стойности в полетата си. Например, нека разгледаме отново класа `Dog`:

```
public class Dog
{
   // Instance variables
   private string name;
   private int age;
}
```

Той има две полета съответно за име – **`name`** и възраст – **`age`**. Във всеки обект, всяко едно от тези полета има собствена стойност, която се съхранява на различно място в паметта за всеки обект.

Понякога обаче, искаме да имаме полета, които са общи за всички обекти от даден клас. За да постигнем това, трябва в декларацията на тези полета да използваме модификатора **`static`**. Както вече обяснихме, такива полета се наричат **статични полета (static fields)**. В литературата се срещат, също и като **променливи на класа**.

Казваме, че статичните полета са **асоциирани с класа**, вместо с който и да е обект от този клас. Това означава, че всички обекти, създадени по описанието на един клас **споделят** статичните полета на класа.

Всички обекти, създадени по описанието на един клас споделят статичните полета на класа.

Декларация на статични полета

Статичните полета декларираме по същия начин, както се декларира поле на клас, като след модификатора за достъп (ако има такъв), добавяме ключовата дума **`static`**:

```
[<access_modifier>] static <field_type> <field_name>
```

Ето как би изглеждало едно поле **`dogCount`**, което пази информация за броя на създадените обекти от клас **`Dog`**:

Dog.cs

```
public class Dog
{
   // Static (class) variable
   static int dogCount;

   // Instance variables
   private string name;
```

```
    private int age;
}
```

Статичните полета се създават, когато за първи път се опитаме да ги достъпим (прочетем / модифицираме). След създаването си, по подобие на обикновените полета в класа, те се инициализират с подразбиращата се стойност за типа си.

Инициализация по време на декларация

Ако по време на декларация на статичното поле, сме задали стойност за инициализация, тя се присвоява на съответното статично поле. Тази инициализация се изпълнява само веднъж – при първото достъпване на полето, веднага след като приключи присвояването на стойността по подразбиране. При последващо достъпване на полето, тази инициализация на статичното поле няма да се изпълни.

В горния пример можем да добавим инициализация на статичното поле:

```
// Static variable - declaration and initialization
static int dogCount = 0;
```

Тази инициализация ще се извърши при първото обръщение към статичното поле. Когато се опитаме да достъпим някое статично поле на класа, ще се задели памет за него и то ще се инициализира със стойностите по подразбиране. След това, ако полето има инициализация по време на декларацията си (както е в нашия случай с полето **dogCount**), тази инициализация ще се извърши. В последствие обаче, когато се опитваме да достъпим полето от други части на програмата ни, този процес няма да се повтори, тъй като статичното поле вече съществува и е инициализирано.

Достъп до статични полета

За разлика от обикновените (нестатични) полета на класа, статичните полета, бидейки асоциирани с класа, а не с конкретен обект, могат да бъдат достъпвани от външен клас като към името на класа, чрез точкова нотация, достъпим името на съответното статично поле:

```
<class_name>.<static_field_name>
```

Например, ако искаме да отпечатаме стойността на статичното поле, което пази броя на създадените обекти от нашия клас **Dog**, това ще стане по следния начин:

```
public static void Main()
{
    // Access to the static variable through class name
    Console.WriteLine("Dog count is now " + Dog.dogCount);
}
```

Съответно, резултатът от изпълнението на този **`Main()`** метод е:

```
Dog count is now 0
```

В C# статичните полета не могат да се достъпват през обект на класа (за разлика от други обектноориентирани езици за програмиране).

Когато даден метод се намира в класа, в който е дефинирано дадено статично поле, то може да бъде достъпено директно без да се задава името на класа, защото то се подразбира:

```
<static_field_name>
```

Модификация на стойностите на статичните полета

Както вече стана дума по-горе, статичните променливи на класа, са споделени от всички обекти и не принадлежат на нито един обект от класа. Съответно, това дава възможност, всеки един от обектите на класа да променя стойностите на статичните полета, като по този начин останалите обекти ще могат да "видят" модифицираната стойност.

Ето защо, например, за да отчетем броя на създадените обекти от клас **`Dog`**, е удобно да използваме статично поле, което увеличаваме с единица, при всяко извикване на конструктора на класа, т.е. всеки път, когато създаваме обект от нашия клас:

```
public Dog(string name, int age)
{
  this.name = name;
  this.age = age;

  // Modifying the static counter in the constructor
  Dog.dogCount += 1;
}
```

Тъй като осъществяваме достъп до статично поле на класа **`Dog`** от него самия, можем да си спестим уточняването на името на класа и да ползваме следния код за достъп до полето **`dogCount`**:

```
public Dog(string name, int age)
{
  this.name = name;
  this.age = age;

  // Modifying the static counter in the constructor
  dogCount += 1;
}
```

За препоръчване е, разбира се, първият начин, при който е очевидно, че полето е статично в класа **`Dog`**. При него кодът е по-лесно четим.

Съответно, за да проверим дали това, което написахме е вярно, ще създадем няколко обекта от нашия клас **`Dog`** и ще отпечатаме броя им. Това ще стане по следния начин:

```
public static void Main()
{
    Dog dog1 = new Dog("Karaman", 1);
    Dog dog2 = new Dog("Bobi", 2);
    Dog dog3 = new Dog("Sharo", 3);

    // Access to the static variable
    Console.WriteLine("Dog count is now " + Dog.dogCount);
}
```

Съответно изходът от изпълнението на примера е:

```
Dog count is now 3
```

Константи (constants)

Преди да приключим с темата за статичните полета, трябва да се запознаем с един по-особен вид статични полета.

По подобие на константите от математиката, в C#, могат да се създадат специални полета на класа, наречени **константи**. Декларирани и инициализирани веднъж константите, винаги притежават една и съща стойност за всички обекти от даден тип.

В C# константите биват два вида:

1. Константи, чиято стойност се извлича по време на компилация на програмата (compile-time константи).
2. Константи, чиято стойност се извлича по време на изпълнение на програмата (run-time константи).

Константи инициализирани по време на компилация (compile-time constants)

Константите, които се изчисляват по време на компилация се декларират по следния начин, използвайки модификатора **`const`**:

```
[<access_modifiers>] const <type> <name>;
```

Константите, декларирани със запазената дума **`const`**, са статични полета. Въпреки това, в декларацията им не се изисква (нито е позволена от компилатора) употребата на модификатора **`static`**:

Въпреки, че константите декларирани с модификатор `const` са статични полета, в декларацията им не трябва и не може да се използва модификаторът `static`.

Например, ако искаме да декларираме като константа числото "пи", познато ни от математиката, това може да стане по следния начин:

```
public const double PI = 3.141592653589793;
```

Стойността, която присвояваме на дадена константа може да бъде израз, който трябва да бъде изчислим от компилатора по време на компилация. Например, както знаем от математиката, константата "пи" може да бъде представена като приблизителен резултат от делението на числата 22 и 7:

```
public const double PI = 22d / 7;
```

При опит за отпечатване на стойността на константата:

```
public static void Main()
{
   Console.WriteLine("The value of PI is: " + PI);
}
```

в командния ред ще бъде изписано:

```
The value of PI is: 3.14285714285714
```

Ако не дадем стойност на дадена константа по време на декларацията й, а по-късно, ще получим грешка при компилация. Например, ако в примера с константата `PI`, първо декларираме константата, и по-късно се опитаме да й дадем стойност:

```
public const double PI;

// ... Some code ...

public void SetPiValue()
{
   // Attempting to initialize the constant PI
   PI = 3.141592653589793;
}
```

Компилаторът ще изведе грешка подобна на следната, указвайки ни реда, на който е декларирана константата:

```
A const field requires a value to be provided
```

Нека обърнем внимание отново:

Константите декларирани с модификатор `const` задължително се инициализират в момента на тяхната декларация.

Тип на константите инициализирани по време на компилация

След като научихме как се декларират константи, които се инициализират по време на компилация, нека разгледаме следния пример: Искаме да създадем клас за цвят (**`Color`**). Ще използваме т.нар. Red-Green-Blue (RGB) цветови модел, съгласно който всеки цвят е представен чрез смесване на трите основни цвята – червен, зелен и син. Тези три основни цвята са представени като три цели числа в интервала от 0 до 255. Например черното е представено като (0, 0, 0), бялото като (255, 255, 255), синьото – (0, 0, 255) и т.н.

В нашия клас декларираме три целочислени полета за червена, зелена и синя светлина и конструктор, който приема стойности за всяко едно от тях:

Color.cs

```
class Color
{
  private int red;
  private int green;
  private int blue;

  public Color(int red, int green, int blue)
  {
    this.red = red;
    this.green = green;
    this.blue = blue;
  }
}
```

Тъй като някои цветове се използват по-често от други (например черно и бяло) можем да декларираме константи за тях, с идеята потребителите на нашия клас да ги използват наготово вместо всеки път да създават свои собствени обекти за въпросните цветове. За целта модифицираме кода на нашия клас по следния начин, добавяйки декларацията на съответните цветове-константи:

Color.cs

```
class Color
{
  public const Color Black = new Color(0, 0, 0);
  public const Color White = new Color(255, 255, 255);
```

```
    private int red;
    private int green;
    private int blue;

    public Color(int red, int green, int blue)
    {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }
}
```

Странно, но при опит за компилация, получаваме следната грешка:

```
'Color.Black' is of type 'Color'. A const field of a reference
type other than string can only be initialized with null.
'Color.White' is of type 'Color'. A const field of a reference
type other than string can only be initialized with null.
```

Това е така, тъй като в C#, константи, декларирани с модификатор **`const`**, могат да бъдат само от следните типове:

1. Примитивни типове: **`sbyte`**, **`byte`**, **`short`**, **`ushort`**, **`int`**, **`uint`**, **`long`**, **`ulong`**, **`char`**, **`float`**, **`double`**, **`decimal`**, **`bool`**.
2. Изброени типове (разгледани в секция "Изброени типове (enumerations)" в края на тази глава).
3. Референтни типове (най-вече типът **`string`**).

Проблемът при компилацията на класа в нашия пример е свързан с референтните типове и с ограничението на компилатора да не позволява едновременната употреба на оператора **`new`** при деклариране на константа, когато тази константа е декларирана с модификатора **`const`**, освен ако референтният тип не може да се изчисли по време на компилация.

Както се досещаме, единственият референтен тип, който може да бъде изчислен по време на компилация при употребата на оператора **`new`** е **`string`**.

Следователно, единствените възможности за константи от референтен тип, които са декларирани с модификатор **`const`**, са следните:

1. Константите трябва да са от тип **`string`**.
2. Стойността, която присвояваме на константата от референтен тип, различен от **`string`**, е **`null`**.

Можем да формулираме следната дефиниция:

Константите декларирани с модификатор `const` трябва да са от примитивен, изброен или референтен тип като ако са от референтен тип, то този тип трябва да е или `string` или стойността, която се присвоява на константата трябва да бъде `null`.

Следователно, използвайки модификатора **`const`** няма да успеем да декларираме константите **`Black`** и **`White`** от тип **`Color`** в нашия клас за цвят, тъй като те не са **`null`**. Как да решим този проблем, ще видим в следващата подсекция.

Константи инициализирани по време на изпълнение на програмата

Когато искаме да декларираме константи от референтен тип, които не могат да бъдат изчислени по време на компилация на програмата, вместо модификатора **`const`**, в декларацията на константата трябва да използваме комбинацията от модификатори **`static readonly`**:

```
[<access_modifiers>] static readonly <reference-type> <name>;
```

Съответно **`<reference-type>`** е такъв тип, чиято стойност не може да бъде изчислена по време на компилация.

Сега, ако заменим **`const`** със **`static readonly`** в последния пример от предходната секция, компилацията минава успешно:

```
public static readonly Color Black = new Color(0, 0, 0);
public static readonly Color White = new Color(255, 255, 255);
```

Именуване на константите

Съгласно конвенцията на Microsoft имената на константите в C# следват правилото **`PascalCase`**. Ако константата е съставена от няколко думи, всяка нова дума след първата започва с главна буква. Ето няколко примера за правилно именувани константи:

```
// The base of the natural logarithms (approximate value)
public const double E = 2.718281828459045;
public const double PI = 3.141592653589793;
public const char PathSeparator = '/';
public const string BigCoffee = "big coffee";
public const int MaxValue = 2147483647;
public static readonly Color DeepSkyBlue = new Color(0,104,139);
```

Понякога за константите се ползва и именуване в стил ALL-CAPS, но то не се подкрепя официално от код конвенциите на Майкрософт, макар и да е силно разпространено в програмирането:

```
public const double FONT_SIZE_IN_POINTS = 14; // 14pt font size
```

Както стана ясно от примерите, разликата между **`const`** и **`static readonly`** полетата е в момента, в който им се присвояват стойностите. Compile-time константите (**`const`**) трябва да бъдат инициализирани в момента на декларацията си, докато run-time константите (**`static readonly`**) могат да бъдат инициализирани на по-късен етап, например в някой от конструкторите на класа, в който са дефинирани.

Употреба на константите

Константите в програмирането се използват, за да се избегне повторението на числа, символни низове или други често срещани стойности (литерали) в програмата и да се позволи тези стойности лесно да се променят. Използването на константи вместо твърдо забити в кода повтарящи се стойности улеснява четимостта и поддръжката на кода и е препоръчителна практика. Според някои автори всички литерали, различни от `0`, `1`, `-1`, празен низ, **`true`**, **`false`** и **`null`** трябва да се декларират като константи, но понякога това затруднява четенето и поддръжката на кода вместо да го опрости. По тази причина се счита, че като константи трябва да се обявят стойностите, които се срещат повече от веднъж в програмата или има вероятност да бъдат променени с течение на времето.

Кога и как да използваме ефективно константите ще научим в подробности в главата "Качествен програмен код".

Статични методи

По подобие на статичните полета, когато искаме един метод да е асоцииран само с класа, но не и с конкретен обект от класа, тогава го декларираме като статичен.

Декларация на статични методи

Синтактично да декларираме **статичен метод** означава, в декларацията на метода, да добавим ключовата дума **`static`**:

```
[<access_modifier>] static <return_type> <method_name>()
```

Нека например декларираме метода за събиране на две числа, за който говорихме в началото на настоящата секция:

```
public static int Add(int number1, int number2)
{
   return (number1 + number2);
}
```

Достъп до статични методи

Както и при статичните полета, статичните методи могат да бъдат достъпвани чрез точкова нотация (операторът точка) приложена към името на класа, като името на класа може да се пропусне ако извикването се извършва от същия клас, в който е деклариран статичният метод. Ето един пример за извикване на статичния метод **`Add(…)`**:

```
public static void Main()
{
   // Call the static method through its class
   int sum = MyMathClass.Add(3, 5);

   Console.WriteLine(sum);
}
```

Достъп между статични и нестатични членове

В повечето случаи статичните методи се използват за достъпване на статични полета от класа, в който са дефинирани. Например, когато искаме да декларираме метод, който да връща броя на създадените обекти от класа **`Dog`**, той трябва да бъде статичен, защото нашият брояч също е статичен:

```
public static int GetDogCount()
{
   return dogCount;
}
```

Но когато разглеждаме как статични и нестатични методи и полета могат да се достъпват, не всички комбинации са позволени.

Достъп до нестатичните членове на класа от нестатичен метод

Нестатичните методи могат да достъпват нестатичните полета и други нестатични методи на класа. Например, в класа **`Dog`** можем да декларираме метод **`PrintInfo()`**, който извежда информация за нашето куче:

Dog.cs

```
public class Dog
{
   // Static variable
   static int dogCount;

   // Instance variables
   private string name;
   private int age;
   public Dog(string name, int age)
   {
      this.name = name;
      this.age = age;

      dogCount += 1;
   }

   public void Bark()
   {
      Console.Write("wow-wow");
   }

   // Non-static (instance) method
   public void PrintInfo()
   {
      // Accessing instance variables – name and age
      Console.Write("Dog's name: " + this.name + "; age: "
         + this.age + "; often says: ");

      // Calling instance method
      this.Bark();
   }
}
```

Разбира се, ако създадем обект от класа **`Dog`** и извикаме неговия **`PrintInfo()`** метод:

```
public static void Main()
{
   Dog dog = new Dog("Sharo", 1);
   dog.PrintInfo();
}
```

Резултатът ще бъде следният:

```
Dog's name: Sharo; age: 1; often says: wow-wow
```

Достъп до статичните елементи на класа от нестатичен метод

От нестатичен метод, можем да достъпваме статични полета и статични методи на класа. Както разбрахме по-рано, това е така, тъй като статичните методи и променливи са обвързани с класа, вместо с конкретен метод и статичните елементи могат да се достъпват от кой да е обект на класа, дори от външни класове (стига да са видими за тях).

Например:

Circle.cs

```
public class Circle
{
   public static double PI = 3.141592653589793;

   private double radius;

   public Circle(double radius)
   {
      this.radius = radius;
   }

   public static double CalculateSurface(double radius)
   {
      return (PI * radius * radius);
   }

   public void PrintSurface()
   {
      double surface = CalculateSurface(radius);
      Console.WriteLine("Circle's surface is: " + surface);
   }
}
```

В примера от нестатичния метод **`PrintSurface()`** осъществяваме достъп до стойността на статичното поле **`PI`**, както извикваме статичния метод **`CalculateSurface()`**. Нека опитаме да извикаме въпросния нестатичен метод:

```
public static void Main()
{
   Circle circle = new Circle(3);
   circle.PrintSurface();
}
```

След компилация и изпълнение, на конзолата ще бъде изведено:

```
Circle's surface is: 28.2743338823081
```

Достъп до статичните елементи на класа от статичен метод

От статичен метод можем да извикваме друг статичен метод или статично поле на класа безпроблемно.

Например, нека вземем нашия клас за математически пресмятания. В него имаме декларирана константата **`PI`**. Можем да декларираме статичен метод за намиране дължината на окръжност (формулата за намиране периметър на окръжност е **2πr**, където **r** е радиусът на окръжността), който за пресмятането на периметъра на дадена окръжност, ползва константата **PI**. След това, за да покажем, че статичен метод може да вика друг статичен метод, можем от статичния метод **Main()** да извикаме статичния метод за намиране периметъра на окръжност:

MyMathClass.cs

```
public class MyMathClass
{
  public const double PI = 3.141592653589793;

  // The method applies the formula: P = 2 * PI * r
  public static double CalculateCirclePerimeter(double r)
  {
    // Accessing the static variable PI from static method
    return (2 * PI * r);
  }

  public static void Main()
  {
    double radius = 5;

    // Accessing static method from other static method
    double circlePerimeter = CalculateCirclePerimeter(radius);

    Console.WriteLine("Circle with radius " + radius +
      " has perimeter: " + circlePerimeter);
  }
}
```

Кодът се компилира без грешки и при изпълнение извежда следния резултат:

```
Circle with radius 5.0 has perimeter: 31.4159265358979
```

Достъп до нестатичните елементи на класа от статичен метод

Нека разгледаме най-интересния случай от комбинацията от достъпване на статични и нестатични елементи на класа – достъпването на нестатични елементи от статичен метод.

Трябва да знаем, че от статичен метод не могат да бъдат достъпвани нестатични полета, нито да бъдат извиквани нестатични методи. Това е така, защото статичните методи са обвързани с класа, и не "знаят" за нито един обект от класа. Затова, ключовата дума **this** не може да се използва в статични методи – тя е обвързана с конкретна инстанция на класа. При опит за достъпване на нестатични елементи на класа (полета или методи) от статичен метод, винаги ще получаваме грешка при компилация.

Непозволен достъп до нестатично поле от статичен метод – пример

Ако в нашия клас **`Dog`** се опитаме да декларираме статичен метод **`PrintName()`**, който връща като резултат стойността на нестатичното поле **`name`** декларирано в класа:

```
public void string PrintName()
{
  // Trying to access non-static variable from static method
  Console.WriteLine(name); // INVALID
}
```

Съответно компилаторът ще ни отговори със съобщение за грешка, подобно на следното:

```
An object reference is required for the non-static field,
method, or property 'Dog.name'
```

Ако въпреки това, се опитаме в метода да достъпим полето чрез ключовата дума **`this`**:

```
public void string PrintName()
{
  // Trying to access non-static variable from static method
  Console.WriteLine(this.name); // INVALID
}
```

Компилаторът отново няма да е доволен и този път ще изведе следното съобщение, без да успее да компилира класа:

```
Keyword 'this' is not valid in a static property, static method,
or static field initializer
```

Непозволено извикване на нестатичен метод от статичен метод – пример

Сега ще се опитаме да извикаме нестатичен метод от статичен метод. Нека в нашия клас **`Dog`** декларираме нестатичен метод **`PrintAge()`**, който отпечатва стойността на полето **`age`**:

```
public void PrintAge()
{
  Console.WriteLine(this.age);
}
```

Съответно, нека се опитаме от метода **`Main()`**, който декларираме в класа **`Dog`**, да извикаме този метод без да създаваме обект от нашия клас:

```
public static void Main()
{
  // Attempt to invoke non-static method from a static context
  PrintAge(); // INVALID
}
```

При опит за компилация ще получим следната грешка:

```
An object reference is required for the non-static field,
method, or property 'Dog.PrintAge()'
```

Резултатът е подобен, ако се опитаме да измамим компилатора, опитвайки се да извикаме метода чрез ключовата дума **`this`**:

```
public static void Main()
{
  // Attempt to invoke non-static method from a static context
  this.PrintAge(); // INVALID
}
```

Съответно, както в случая с опита за достъп до нестатично поле от статичен метод, чрез ключовата дума **`this`**, компилаторът извежда следното съобщение, без да успее да компилира нашия клас:

```
Keyword 'this' is not valid in a static property, static method,
or static field initializer
```

От разгледаните примери, можем да направим следния извод:

Нестатичните елементи на класа НЕ могат да бъдат използвани в статичен контекст.

Проблемът с достъпа до нестатични елементи на класа от статичен метод има едно единствено решение – тези нестатични елементи да се достъпват чрез референция към даден обект:

```
public static void Main()
{
  Dog myDog = new Dog("Sharo", 2);
  string myDogName = myDog.name;
  Console.WriteLine("My dog \"" + myDogName + "\" has age of ");
  myDog.PrintAge();
  Console.WriteLine("years");
}
```

Съответно този код се компилира и резултатът от изпълнението му е:

```
My dog "Sharo" has age of 2 years
```

Статични свойства на класа

Макар и рядко, понякога е удобно да се декларират и използват свойства не на обекта, а на класа. Те носят същите характеристики като свойствата, свързани с конкретен обект от даден клас, които разгледахме по-горе, но с тази разлика, че статичните свойства се отнасят за класа.

Както можем да се досетим, всичко, което е нужно да направим, за да превърнем едно обикновено свойство в статично, е да добавим ключовата ма **`static`** при декларацията му.

Статичните свойства се декларират по следния начин:

```
[<modifiers>] static <property_type> <property_name>
{
  // ... Property’s accessors methods go here
}
```

Нека разгледаме един пример. Имаме клас, който описва някаква система. Можем да създаваме много обекти от нея, но моделът на системата има дадена версия и производител, които са общи за всички екземпляри, създадени от този клас. Можем да направим версията и производителите статични свойства на класа:

SystemInfo.cs

```
public class SystemInfo
{
  private static double version = 0.1;
  private static string vendor = "Microsoft";

  // The "version" static property
  public static double Version
  {
    get { return version; }
    set { version = value; }
  }

  // The "vendor" static property
  public static string Vendor
  {
    get { return vendor; }
    set { vendor = value; }
  }

  // ... More (non)static code here ...
}
```

В този пример сме избрали да пазим стойността на статичните свойства в статични променливи (което е логично, тъй като те са обвързани само с класа). Свойствата, които разглеждаме са съответно версия (**`Version`**) и производител (**`Vendor`**). За всяко едно от тях сме създали статични методи за четене и модификация. Така всички обекти от този клас, ще могат да извлекат текущата версия и производителя на системата, която описва класа. Съответно, ако някой ден бъде направено обновление на версията на системата например стойността стане `0.2`, всеки от обектите, ще получи като резултат новата версия, чрез достъпване на свойството на класа.

Статичните свойства и ключовата дума this

Подобно на статичните методи, в статичните свойства не може да се използва ключовата дума **`this`**, тъй като статичното свойство е асоциирано единствено с класа, и не "разпознава" обектите от даден клас.

В статичните свойства не може да се използва ключовата дума this.

Достъп до статични свойства

По подобие на статичните полета и методи, статичните свойства могат да бъдат достъпвани чрез точкова нотация приложена единствено към името на класа, в който са декларирани.

За да се уверим, нека се опитаме да достъпим свойството **`Version`** през променлива от класа **`SystemInfo`**:

```
public static void Main()
{
   SystemInfo sysInfoInstance = new SystemInfo();
   Console.WriteLine("System version: " +
      sysInfoInstance.Version);
}
```

При опит за компилация на горния код, получаваме следното съобщение за грешка:

```
Member 'SystemInfo.Version.get' cannot be accessed with an
instance reference; qualify it with a type name instead
```

Съответно, ако се опитаме да достъпим статичните свойства чрез името на класа, кодът се компилира и работи правилно:

```
public static void Main()
{
   // Invocation of static property setter
   SystemInfo.Vendor = "Microsoft Corporation";

   // Invocation of static property getters
   Console.WriteLine("System version: " + SystemInfo.Version);
```

```
  Console.WriteLine("System vendor: " + SystemInfo.Vendor);
}
```

Кодът се компилира и резултатът от изпълнението му е:

```
System version: 0.1
System vendor: Microsoft Corporation
```

Преди да преминем към следващата секция, нека обърнем внимание на отпечатаната стойност на свойството `Vendor`. Тя е "`Microsoft Corporation`", въпреки че в класа `SystemInfo` сме я инициализирали със стойността "`Microsoft`". Това е така, тъй като променихме стойността на свойството `Vendor` на първия ред от метода `Main()`, чрез извикване на метода му за модификация.

Статичните свойства могат да бъдат достъпвани единствено чрез точкова нотация, приложена към името на класа, в който са декларирани.

Статични класове

За пълнота трябва да обясним, че можем да декларираме класовете като статични. Подобно на статичните членове, един клас е статичен, когато при декларацията му е използвана ключовата дума `static`:

```
[<modifiers>] static class <class_name>
{
  // ... Class body goes here
}
```

Когато един клас е деклариран като статичен, това е индикация, че този клас съдържа само статични членове (т.е. статични полета, методи, свойства) и не може да се инстанцира.

Употребата на статични класове е рядка и най-често е свързана с употребата на статични методи, които не принадлежат на нито един конкретен обект. По тази причина, подробностите за статичните класове излизат извън обсега на тази книга. Любознателният читател може да намери повече информация на сайта на Microsoft (MSDN).

Статични конструктори

За да приключим със секцията за статичните членове на класа, трябва да споменем и класовете могат да имат и **статичен конструктор** (т.е. конструктор, които има ключовата дума `static` в декларацията си):

```
[<modifiers>] static <class_name>([<parameters_list>])
{
}
```

Статични конструктори могат да бъдат декларирани, както в статични, така и в нестатични класове. Те се изпълняват само веднъж, когато първото от следните две събития се случи за първи път:

1. Създава се обект от класа.
2. Достъпен е статичен елемент от класа (поле, метод, свойство).

Най-често статичните конструктори се използват за инициализацията на статични полета.

Статичен конструктор – пример

Да разгледаме един пример за използването на статичен конструктор. Искаме да направим клас, който изчислява бързо корен квадратен от цяло число и връща цялата част на резултата – също цяло число. Тъй като изчисляването на корен квадратен е времеотнемаща математическа операция, включваща пресмятания с реални числа и изчисляване на сходящи редове, е добра идея тези изчисления да се изпълнят еднократно при стартиране на програмата, а след това да се използват вече изчислени стойности. Разбира се, за да се направи такова предварително изчисление (precomputation) на всички квадратни корени в даден диапазон, трябва първо да се дефинира този диапазон и той не трябва да е прекалено широк (например от 1 до 1000). След това е необходимо при първо поискване на корен квадратен на дадено число да се преизчислят всички квадратни корени в дадения диапазон, а след това да се върне вече готовата изчислена стойност. При следващо поискване на корен квадратен, всички стойности в дадения диапазон са вече изчислени и се връщат директно. Ако пък никога в програмата не се изчислява корен квадратен, предварителните изчисления трябва изобщо да не се изпълнят.

Чрез описания подход първоначално се инвестира някакво процесорно време за предварителни изчисления, но след това извличането на корен квадратен се извършва изключително бързо. Ако изчисляването на корен квадратен се извършва многократно, преизчислението ще увеличи значително производителността.

Всичко това може да се имплементира в един статичен клас със статичен конструктор, в който да се преизчисляват квадратните корени. Вече изчислените резултати могат да се съхраняват в статичен масив. За извличане на вече преизчислена стойност може да се използва статичен метод. Тъй като предварителните изчисления се извършват в статичния конструктор, ако класът за преизчислен корен квадратен не се използва, те няма да се извършат и ще се спести процесорно време и памет. Ето как би могла да изглежда имплементацията:

```
static class SqrtPrecalculated
{
   public const int MaxValue = 1000;

   // Static field
   private static int[] sqrtValues;

   // Static constructor
   static SqrtPrecalculated()
   {
      sqrtValues = new int[MaxValue + 1];
      for (int i = 0; i < sqrtValues.Length; i++)
      {
         sqrtValues[i] = (int)Math.Sqrt(i);
      }
   }

   // Static method
   public static int GetSqrt(int value)
   {
      if ((value < 0) || (value > MaxValue))
      {
         throw new ArgumentOutOfRangeException(String.Format(
            "The argument should be in range [0..{0}].",
            MaxValue));
      }
      return sqrtValues[value];
   }
}

class SqrtTest
{
   static void Main()
   {
      Console.WriteLine(SqrtPrecalculated.GetSqrt(254));
      // Result: 15
   }
}
```

Изброени типове (enumerations)

По-рано в тази глава ние разгледахме какво представляват константите, как се декларират и как се използват. В тази връзка, сега ще разгледаме една конструкция от езика C#, при която можем множество от константи, които са свързани логически, да ги свържем и чрез средствата на езика. Това средство на езика са така наречените изброени типове.

Декларация на изброените типове

Изброен тип (enumeration) наричаме конструкция, която наподобява клас, но с тази разлика, че в тялото на класа можем да декларираме само константи. Изброените типове могат да приемат стойности само измежду изброените в типа константи. Променлива от изброен тип може да има за стойност някоя измежду изброените в типа стойности (константи), но не може да има стойност `null`.

Формално казано, изброените типове се декларират с помощта на запазената дума `enum` вместо `class`:

```
[<modifiers>] enum <enum_name>
{
  constant1 [, constant2 [, [, ... [, constantN]]
}
```

Под `<modifiers>` разбираме модификаторите за достъп `public`, `internal` и `private`. Идентификаторът `<enum_name>` следва правилата за имена на класове в C#. В блока на изброения тип се декларират константите, разделени със запетайки.

Нека разгледаме един пример. Да дефинираме изброен тип за дните от седмицата (ще го наречем `Days`). Както се досещаме, константите, които ще се съдържат в този изброен тип са имената на дните от седмицата:

Days.cs

```
enum Days
{
  Mon, Tue, Wed, Thu, Fri, Sat, Sun
}
```

Именуването на константите в един изброен тип следва правилото за именуване на константи, което обяснихме в секцията "Именуване на константите".

Трябва да отбележим, че всяка една от константите в изброения тип е от тип този изброен тип, т.е. в нашия пример `Mon` е от тип `Days`, както и всяка една от останалите константи.

С други думи, ако изпълним следния ред:

```
Console.WriteLine(Days.Mon is Days);
```

ще бъде отпечатан резултат:

```
True
```

Нека повторим:

Изброените типове са множество от константи от тип – този изброен тип.

Същност на изброените типове

Всяка една константа, която е декларирана в един изброен тип, е асоциирана с някакво цяло число. По подразбиране, за това целочислено скрито представяне на константите в един изброен тип се използва **`int`**.

За да покажем "целочислената природа" на константите в изброените типове, нека се опитаме да разберем какво е численото представяне на константата отговаряща на "понеделник" от примера от предходната подсекция:

```
int mondayValue = (int)Days.Mon;
Console.WriteLine(mondayValue);
```

След като го изпълним, резултатът ще бъде:

```
0
```

Стойностите, асоциирани с константите в един изброен тип по подразбиране са индексите в списъка с константи на този тип, т.е. числата от 0 до броя константи в типа минус единица. Така, ако разгледаме примера с изброения тип за дните в седмицата, използван в предходната подсекция, константата **`Mon`** е асоциирана с числената стойност 0, константата **`Tue`** с целочислената стойност 1, **`Wed`** – с 2, и т.н.

Всяка константа в един изброен тип реално е текстово представяне на някакво цяло число. По подразбиране, това число е индексът на константата в списъка от константи на изброения тип.

Въпреки целочислената природа на константите в един изброен тип, когато се опитаме да отпечатаме дадена константа, ще бъде отпечатано текстовото ѝ представяне зададено при декларацията ѝ:

```
Console.WriteLine(Days.Mon);
```

След като изпълним горния код, резултатът ще бъде следният:

```
Mon
```

Скрита числена стойност на константите в изброени типове

Както вече се досещаме, можем да променим **числената стойност на константите в един изброен тип**. Това става като по време на декларацията присвоим стойността, която предпочитаме, на всяка една от константите.

```
[<modifiers>] enum <enum_name>
{
   constant1[=value1] [, constant2[=value2] [, ... ]]
}
```

Съответно **`value1`**, **`value2`**, и т.н. трябва да са цели числа.

За да добием по-ясна представа за току-що дадената дефиниция, нека разгледаме следния пример: нека имаме клас **`Coffee`**, който представя чаша кафе, която клиентите поръчват в някакво заведение:

Coffee.cs

```
public class Coffee
{
   public Coffee()
   {
   }
}
```

В това заведение, клиентът може да поръча различно количество кафе, като кафе-машината има предефинирани стойности – "малко" – 100 ml, "нормално" – 150 ml и "двойно" – 300 ml. Следователно, можем да си декларираме един изброен тип **`CoffeSize`**, който има съответно три константи – **`Small`**, **`Normal`** и **`Double`**, на които ще присвоим съответстващите им количества:

CoffeeSize.cs

```
public enum CoffeeSize
{
   Small=100, Normal=150, Double=300
}
```

Сега можем да добавим поле и свойство към класа **`Coffee`**, които отразяват какъв тип кафе си е поръчал даден клиент:

Coffee.cs

```
public class Coffee
{
    public CoffeeSize size;

    public Coffee(CoffeeSize size)
    {
        this.size = size;
    }

    public CoffeeSize Size
    {
        get { return size; }
    }
}
```

Нека се опитаме да отпечатаме стойностите на количеството кафе за едно нормално кафе и за едно двойно:

```
static void Main()
{
    Coffee normalCoffee = new Coffee(CoffeeSize.Normal);
    Coffee doubleCoffee = new Coffee(CoffeeSize.Double);

    Console.WriteLine("The {0} coffee is {1} ml.",
        normalCoffee.Size, (int)normalCoffee.Size);
    Console.WriteLine("The {0} coffee is {1} ml.",
        doubleCoffee.Size, (int)doubleCoffee.Size);
}
```

Както компилираме и изпълним този метод, ще бъде отпечатано следното:

```
The Normal coffee is 150 ml.
The Double coffee is 300 ml.
```

Употреба на изброените типове

Основната цел на изброените типове е да заменят числените стойности, които бихме използвали, ако не съществуваха изброените типове. По този начин, кодът става по-изчистен и по-лесен за четене.

Друго много важно приложение на изброените типове е принудата от страна на компилатора да бъдат използвани константите от изброения тип, а не просто числа. По този начин ограничаваме максимално бъдещи грешки в кода. Например, ако използваме променлива от тип `int` вместо от изброен тип и набор константи за валидните стойности, нищо не пречи да присвоим на променливата примерно -6723.

За да стане по-ясно, нека разгледаме следния пример: да създадем клас, който представлява калкулатор за пресмятане на цената на всеки от видовете кафе, които се предлагат в заведението:

PriceCalculator.cs

```
public class PriceCalculator
{
  public const int SmallCoffeeQuantity = 100;
  public const int NormalCoffeeQuantity = 150;
  public const int DoubleCoffeeQuantity = 300;

  public CashMachine() { }

  public double CalcPrice(int quantity)
  {
    switch (quantity)
    {
      case SmallCoffeeQuantity:
        return 0.20;
      case NormalCoffeeQuantity:
        return 0.30;
      case DoubleCoffeeQuantity:
        return 0.60;
      default:
        throw new InvalidOperationException(
          "Unsupported coffee quantity: " + quantity);
    }
  }
}
```

Създали сме три константи, отразяващи вместимостта на чашките за кафе, които имаме в заведението, съответно 100, 150 и 300 ml. Освен това **очакваме**, че потребителите на нашия клас ще използват прилежно дефинираните от нас константи, вместо числа – `SmallCoffeeQuantity`, `NormalCoffeeQuantity` и `DoubleCoffeeQuantity`. Методът `CalcPrice(int)` връща съответната цена, като я изчислява според подаденото количество.

Проблемът, се състои в това, че някой може да реши да не използва дефинираните от нас константи и може да подаде като параметър на нашия метод невалидно число, например -1 или 101. В този случай, ако методът не прави проверка за невалидно количество, най-вероятно ще върне грешна цена, което е некоректно поведение.

За да избегнем този проблем, ще използваме една особеност на изброените типове, а именно, че константите в изброените типове могат да се използват в конструкции `switch-case`. Те могат да бъдат подавани като стойност на оператора `switch` и съответно – като операнди на оператора `case`.

Константите на един изброен тип могат да бъдат използвани в конструкции switch-case.

Нека преработим метода за получаване на цената за чашка кафе в зависимост от вместимостта на чашката, като този път използваме изброения тип **CoffeeSize**, който декларирахме в предходните примери:

```
public double getPrice(CoffeeSize coffeeSize)
{
    switch (coffeeSize)
    {
        case CoffeeSize.Small:
            return 0.20;
        case CoffeeSize.Normal:
            return 0.40;
        case CoffeeSize.Double:
            return 0.60;
        default:
            throw new InvalidOperationException(
                "Unsupported coffee quantity: " +((int)coffeeSize));
    }
}
```

Както виждаме, в този пример възможността потребителите на нашия метод да провокират непредвидено поведение на метода е нищожна, тъй като ги принуждаваме да използват точно определени стойности, които да подадат като аргументи, а именно константите на изброения тип **CoffeeSize**. Това е едно от предимствата на константите, декларирани в изброени типове пред константите декларирани в произволен клас.

Винаги, когато съществува възможност, използвайте изброен тип вместо множество константи декларирани в някакъв клас.

Преди да приключим секцията за изброените типове, трябва да споменем, че изброените типове трябва да се използват много внимателно при работа с конструкцията **switch-case**. Например, ако някой ден, собственикът на заведението купи много големи чаши за кафе, ще трябва да добавим нова константа в списъка с константи на изброения тип **CoffeeSize**, нека я наречем **Overwhelming**:

CoffeeSize.cs

```
public enum CoffeeSize
{
    Small=100, Normal=150, Double=300, Overwhelming=600
}
```

Когато се опитаме да пресметнем цената на кафе с новото количество, методът, който пресмята цената, ще хвърли изключение, което съобщава на потребителя, че такова количество кафе не се предлага в заведението.

Това, което трябва да направим, за да решим този проблем е да добавим ново `case`-условие, което да отразява новата константа в изброения тип `CoffeeSize`.

> **Когато модифицираме списъка с константите на вече съществуващ изброен тип, трябва да внимаваме, да не нарушим логиката на кода, който вече съществува и използва декларираните до момента константи.**

Вътрешни класове (nested classes)

В C# **вътрешен (nested)** се нарича клас, който е деклариран вътре в тялото на друг клас. Съответно, клас, който обвива вътрешен клас се нарича **външен клас (outer class)**.

Основните причини да се декларира един клас в друг са следните:

1. За по-добра организация на кода, когато работим с обекти от реалния свят, между които има специална връзка и единият не може да съществува без другия.
2. Скриване на даден клас в друг клас, така че вътрешният клас да не бъде използван извън обвиващия го клас.

По принцип вътрешни класове се ползват рядко, тъй като те усложняват структурата на кода и увеличават нивата на влагане.

Декларация на вътрешни класове

Вътрешните класове се декларират по същия начин, както нормалните класове, но се разполагат вътре в друг клас. Позволените модификатори в декларацията на класа са следните:

1. `public` – вътрешният клас е достъпен от кое да е асембли.
2. `internal` – вътрешният клас е достъпен в текущото асембли, в което се намира външния клас.
3. `private` – достъпът е ограничен само до класа, който съдържа вътрешния клас.
4. `static` – вътрешният клас съдържа само статични членове.

Има още четири позволени модификатора – `abstract`, `protected`, `protected internal`, `sealed` и `unsafe`, които са извън обхвата и тематиката на тази глава и няма да бъдат разглеждани тук.

Ключовата дума `this` за един вътрешен клас, притежава връзка единствено към вътрешния клас, но не и към външния. Полетата на външния клас **не могат** да бъдат достъпвани използвайки референцията `this`. Ако е необходимо полетата на външния клас да бъдат достъпвани от вътрешния, трябва при създаването на вътрешния клас да се подаде референция към външния клас.

Статичните членове (полета, методи, свойства) на външния клас са достъпни от вътрешния независимо от нивото си на достъп.

Вътрешни класове – пример

Нека разгледаме следния пример:

OuterClass.cs

```
public class OuterClass
{
  private string name;

  private OuterClass(string name)
  {
    this.name = name;
  }

  private class NestedClass
  {
    private string name;
    private OuterClass parent;

    public InnerClass(OuterClass parent, string name)
    {
      this.parent = parent;
      this.name = name;
    }

    public void PrintNames()
    {
      Console.WriteLine("Nested name: " + this.name);
      Console.WriteLine("Outer name: " + this.parent.name);
    }
  }

  public static void Main()
  {
    OuterClass outerClass = new OuterClass("outer");
    NestedClass nestedClass = new
      OuterClass.InnerClass(outerClass, "nested");
    nestedClass.PrintNames();
  }
}
```

В примера външният клас **`OuterClass`** дефинира в себе си като член класа **`InnerClass`**. Нестатичните методи на вътрешния клас имат достъп както до собствената си инстанция **`this`**, така и до инстанцията на външния клас

parent (чрез синтаксиса **this.parent**, ако референцията **parent** е добавена от програмиста). В примера при създаването на вътрешния клас на конструктора му се подава **parent** референцията на външния клас.

Ако изпълним горния пример, ще получим следния резултат:

```
Inner name: inner
Outer name: outer
```

Употреба на вътрешни класове

Нека разгледаме един пример. Нека имаме клас за кола – **Car**. Всяка кола има двигател, както и врати. За разлика от вратите на колата обаче, двигателят няма смисъл разглеждан като елемент извън колата, тъй като без него, колата не може да работи, т.е. имаме композиция (вж. секция "Композиция" в глава "Принципи на обектно-ориентираното програмиране").

Когато връзката между два класа е композиция, класът, който логически е част от друг клас, е удобно да бъде деклариран като вътрешен клас.

Следователно, ако декларираме класа за кола **Car**, ще е подходящо да си създадем като вътрешен клас **Engine**, който ще отразява съответно концепцията за двигател на колата:

Car.cs

```
class Car
{
   Door FrontRightDoor;
   Door FrontLeftDoor;
   Door RearRightDoor;
   Door RearLeftDoor;
   Engine engine;

   public Car()
   {
      engine = new Engine();
      engine.horsePower = 2000;
   }

   public class Engine
   {
      public int horsePower;
   }
}
```

Декларация на изброен тип в клас

Преди да преминем към следващата тема за шаблонните типове (generics), трябва да отбележим, че понякога изброените типове се налага и **могат** да бъдат декларирани в рамките на даден клас с оглед на по-добрата капсулация на класа.

Например, изброеният тип **`CoffeeSize`**, който създадохме в предходната секция може да бъде деклариран вътре в тялото на класа **`Coffee`**, като по този начин се подобрява капсулацията:

Coffee.cs

```
class Coffee
{
   // Enumeration
   public static enum CoffeeSize
   {
      Small = 100, Normal = 150, Double = 300
   }

   // Instance variable
   private CoffeeSize size;

   public Coffee(CoffeeSize size)
   {
      this.size = size;
   }

   public CoffeeSize Size
   {
      get { return size; }
   }
}
```

Съответно, методът за изчисляване на цената на едно кафе ще претърпи лека модификация:

```
public double CalcPrice(Coffee.CoffeeSize coffeeSize)
{
   switch (coffeeSize)
   {
      case Coffee.CoffeeSize.Small:
         return 0.20;
      case Coffee.CoffeeSize.Normal:
         return 0.40;
      case Coffee.CoffeeSize.Double:
         return 0.60;
      default:
         throw new InvalidOperationException(
            "Unsupported coffee quantity: " + ((int)coffeeSize));
    }
}
```

Шаблонни типове и типизиране (generics)

В тази секция ще обясним концепцията за типизиране на класове. Преди да започнем, обаче, нека разгледаме един пример, който ще ни помогне за разберем по-лесно идеята.

Приют за бездомни животни – пример

Нека имаме два класа. Нека класът **`Dog`** описва куче:

Dog.cs

```
public class Dog
{
}
```

И нека класът **`Cat`** описва котка:

Cat.cs

```
public class Cat
{
}
```

След това искаме да си създадем клас, който описва приют за бездомни животни – **`AnimalShelter`**. Този клас има определен брой свободни клетки, който определя броя на животни, които могат да намерят подслон в приюта. Особеното на класа, който искаме да създадем е, че той трябва да подслонява само животни от един и същ вид, в нашия случай или само кучета, или само котки, защото съвместното съжителство на различни видове животни не винаги е добра идея.

Ако се замислим как ще решим задачата със знанията, които имаме до момента, стигаме до извода, че за да гарантираме, че нашият клас ще съдържа елементи само от един тип, трябва да използваме масив от еднакви обекти. Тези обекти може да са кучета, котки или просто инстанции на универсалния тип **`object`**.

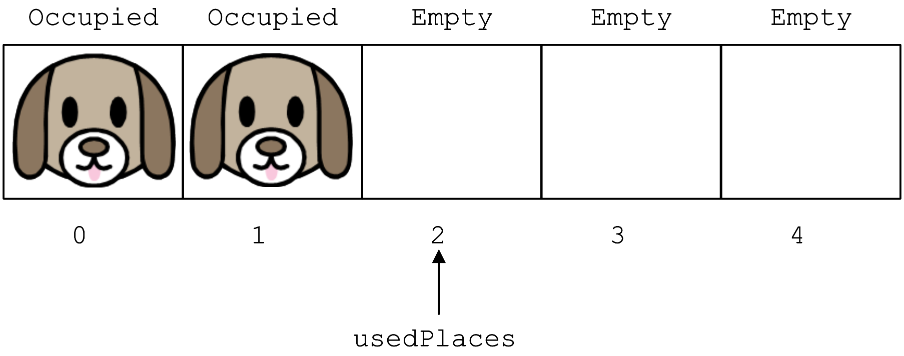
Например, ако искаме да направим приют за кучета, ето как би изглеждал нашият клас:

AnimalsShelter.cs

```
public class AnimalShelter
```

```
{
  private const int DefaultPlacesCount = 20;

  private Dog[] animalList;
  private int usedPlaces;

  public AnimalShelter() : this(DefaultPlacesCount)
  {
  }

  public AnimalShelter(int placesCount)
  {
    this.animalList = new Dog[placesCount];
    this.usedPlaces = 0;
  }

  public void Shelter(Dog newAnimal)
  {
    if (this.usedPlaces >= this.animalList.Length)
    {
      throw new InvalidOperationException("Shelter is full.");
    }
    this.animalList[this.usedPlaces] = newAnimal;
    this.usedPlaces++;
  }

  public Dog Release(int index)
  {
    if (index < 0 || index >= this.usedPlaces)
    {
      throw new ArgumentOutOfRangeException(
        "Invalid cell index: " + index);
    }
    Dog releasedAnimal = this.animalList[index];
    for (int i = index; i < this.usedPlaces - 1; i++)
    {
      this.animalList[i] = this.animalList[i + 1];
    }
    this.animalList[this.usedPlaces - 1] = null;
    this.usedPlaces--;

    return releasedAnimal;
  }
}
```

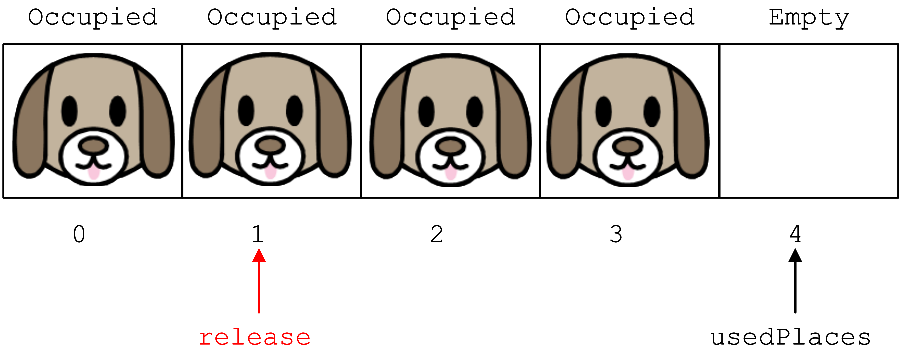
Капацитетът на приюта (броят животни, които могат да се приютят в него) се задава при създаване на обекта. По подразбиране е стойността на константата **DefaultPlacesCount**. Полето **usedPlaces** използваме за следене на заетите до момента клетки (едновременно с това го използваме за индекс в масива, да "сочим" към първото свободно място отляво на дясно в масива).


Occupied
Occupied
Empty
Empty
Empty
0
1
2
3
4
usedPlaces

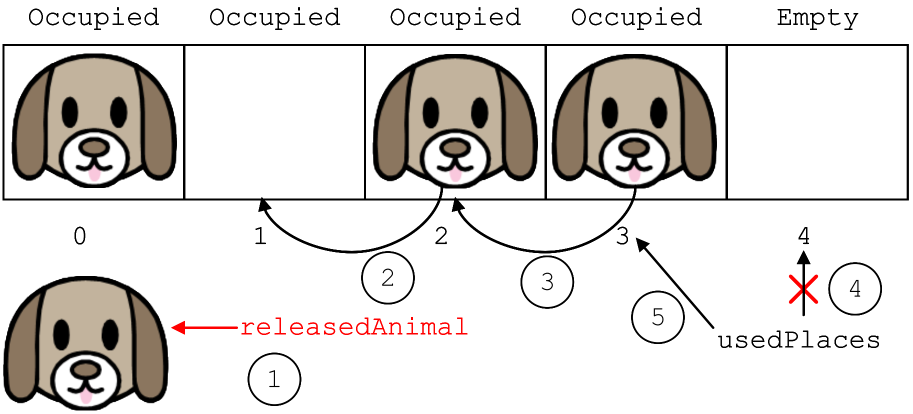
Създали сме метод за добавяне на ново куче в приюта – **`Shelter(…)`** и съответно за освобождаване от приюта – **`Release(int)`**.

Методът **`Shelter()`** добавя всяко ново животно в първата свободна клетка в дясната част на масива (ако има такава).

Методът **`Release(int)`** приема номера на клетката, от която ще бъде освободено куче (т.е. номера на индекса в масива, където е съхранена връзка към обекта от тип **`Dog`**).

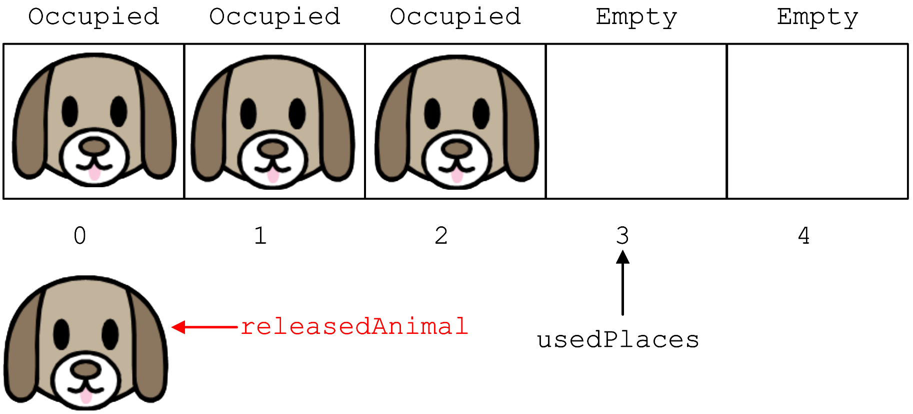

Occupied
Occupied
Occupied
Occupied
Empty
0
1
2
3
4
release
usedPlaces

След това премества животните, които се намират в клетки с по-голям номер от номера на клетката, от която ще извадим едно куче, с една позиция на наляво (стъпки 2 и 3 на схемата по-долу).


Occupied
Occupied
Occupied
Occupied
Empty
0
1
2
3
4
2
3
5
4
releasedAnimal
1
usedPlaces

Освободената клетка на позиция **`usedPlaces-1`** се маркира като свободна, като й се присвоява стойност **`null`**. Това осигурява освобождаването на референцията към нея и съответно позволява на системата за почистване на паметта (garbage collector) да освободи обекта, ако той не се ползва никъде другаде в програмата в същия момент. Това предпазва недиректна загуба на памет (memory leak).

Накрая присвоява на полето **`usedPlaces`** номера на последната свободна клетка (стъпки 4 и 5 от схемата отгоре).


Occupied
Occupied
Occupied
Empty
Empty
0
1
2
3
4
releasedAnimal
usedPlaces

Забелязва се, че "изваждането" на животно от дадена клетка би могло да е бавна операция, тъй като изисква прехвърляне на животните от следващите клетки с една позиция наляво. В главата "Линейни структури от данни" ще разгледаме и по-ефективни начини за представяне на приюта за животни, но за момента нека се фокусираме върху темата за шаблонните типове.

До този момент успяхме да имплементираме функционалността на приюта - класът **`AnimalShelter`**. Когато работим с обекти от тип **`Dog`**, всичко се компилира и изпълнява безпроблемно:

```
public static void Main()
{
   AnimalShelter dogsShelter = new AnimalShelter(10);
   Dog dog1 = new Dog();
   Dog dog2 = new Dog();
   Dog dog3 = new Dog();

   dogsShelter.Shelter(dog1);
   dogsShelter.Shelter(dog2);
   dogsShelter.Shelter(dog3);

   dogsShelter.Release(1); // Releasing dog2
}
```

Какво ще стане, обаче, ако се опитаме да използваме класа **`AnimalShelter`** за обекти от тип **`Cat`**:

```
public static void Main()
{
   AnimalShelter dogsShelter = new AnimalShelter(10);

   Cat cat1 = new Cat();

   dogsShelter.Shelter(cat1);
}
```

Както се очаква, компилаторът извежда грешка:

```
The best overloaded method match for 'AnimalShelter.Shelter(
Dog)' has some invalid arguments. Argument 1: cannot convert
from 'Cat' to 'Dog'
```

Следователно, ако искаме да направим приют за котки, няма да успеем да преизползваме вече създадения от нас клас, въпреки, че операциите по добавяне и изваждане на животни от приюта ще бъдат идентични. Следователно, буквално ще трябва да копираме класа **`AnimalShelter`** и да променим само типа на обектите, с които се работи – **`Cat`**.

Да, но ако решим да правим приют и за други видове животни? Колко класа за приюти за конкретния тип животни ще трябва да създадем?

Виждаме, че това решение на задачата не е достатъчно изчерпателно и не изпълнява изцяло условията, които си бяхме поставили, а именно – да съществува **един единствен клас**, който описва нашия приют за каквито и да е животни (т.е. за всякакви обекти) и при работа с него той да **съдържа само един вид животни** (т.е. единствено обекти от един и същ тип).

Бихме могли да използваме вместо типа **`Dog`** универсалния тип **`object`**, който може да приема като стойности **`Dog`**, **`Cat`** и всякакви други типове данни, но това ще създаде някои неудобства, свързани с нуждата от обратно преобразуване от **`object`** към **`Dog`**, когато се прави приют за кучета, а той съдържа клетки от тип object вместо от тип **`Dog`**.

За да решим задачата ефективно се налага да използваме една функционалност на езика C#, която ни позволява да удовлетворим всички условия едновременно. Тя се нарича **шаблонни класове (generics)**.

Какво представляват шаблонните класове?

Както знаем, когато за работата на един метод е нужна допълнителна информация, тази информация се подава на метода чрез параметри. По време на изпълнение на програмата, при извикване на метода, подаваме аргументи на метода, те се присвояват на параметрите му и след това се използват в тялото на метода.

По подобие на методите, когато знаем, че функционалността (действията) капсулирана в един клас, може да бъде приложена не само към обекти от един, а от много (разнородни) типове, и тези типове не са известни по време на деклариране на класа, можем да използваме една функционалност на езика C# наречена **шаблонни типове (generics)**. Тя ни позволява да декларираме параметри на самия клас, чрез които обозначаваме неизвестния тип, с който класът ще работи в последствие. След това, когато инстанцираме нашия типизиран клас, ние заместваме неизвестния тип с конкретен. Съответно новосъздаденият обект ще работи само с обекти от конкретния тип, който сме задали при инициализацията му. Конкретният тип може да бъде всеки един клас, който компилаторът разпознава, включително структура, изброен тип или друг шаблонен клас.

За да добием по-ясна представа за същността на шаблонните типове, нека се върнем към нашата задача от предходната секция. Както се досещаме, класът, който описва приют на животни (**`AnimalShelter`**), може да оперира с различни типове животни. Следователно, ако искаме да създадем генерално решение на задачата, по време на декларация на класа **`AnimalShelter`**, ние не можем да знаем какъв тип животни ще бъдат приютявани в приюта. Това е достатъчна индикация, че можем да типизираме нашия клас, добавяйки към декларацията на класа, като параметър, неизвестния ни тип на животни.

В последствие, когато искаме да създадем приют за кучета например, на този параметър на класа ще подадем името на нашия тип – класа **`Dog`**. Съответно, ако създаваме приют за котки, ще подадем типа **`Cat`** и т.н.

Типизирането на клас (създаването на шаблонен клас) представлява добавяне, към декларацията на един клас, на параметър (заместител) на неизвестен тип, с който класът ще работи по време на изпълнение на програмата. В последствие, когато класът бива инстанциран, този параметър се замества с името на някой конкретен тип.

В следващите секции ще се запознаем със синтаксиса на типизирането на класове и ще представим нашия пример преработен, така че да използва типизиране.

Декларация на типизиран (шаблонен) клас

Формално, типизирането на класове се прави, като към декларацията на класа, след самото име на класа се добави **`<T>`**, където **`T`** е заместителят (параметърът) на типа, който ще се използва в последствие:

```
[<modifiers>] class <class_name><T>
{
}
```

Трябва да отбележим, че знаците '<' и '>', които ограждат заместителя **`T`** са задължителна част от синтаксиса на езика C# и трябва да участват в декларацията на типизирането на даден клас.

Декларацията на типизирания клас, описващ приюта за бездомни животни, би изглеждала по следния начин:

```
class AnimalShelter<T>
{
   // Class body here ...
}
```

По този начин, можем да си представим, че правим шаблон на нашия клас **`AnimalShelter`**, който в последствие ще конкретизираме, заменяйки **`T`** с конкретен тип, например **`Dog`**.

Един клас може да има и повече от един заместител (да е параметризиран по повече от един тип), в зависимост от нуждите му:

```
[<modifiers>] class <class_name><T1 [, T2, [... [, Tn]>
{
}
```

Ако класът се нуждае от няколко различни неизвестни типа, тези типове трябва да се изброят, чрез запетайка между знаците '<' и '>' в декларацията на класа, като всеки един от използваните заместители трябва да е различен идентификатор (например различна буква) – в дефиницията са указани като **`T1`**, **`T2`**, ..., **`Tn`**.

В случай, че искахме да създадем приют за животни от смесен тип, такъв че да приютява кучета и котки едновременно, можехме да декларираме нашия клас по следния начин:

```
class AnimalShelter<T, U>
{
   // Class body here ...
}
```

Ако това беше нашия случай, щяхме да използваме първия параметър **`T`**, за означаване на обектите от тип **`Dog`**, с които нашия клас щеше да оперира и **`U`** – за означаване на обектите от тип **`Cat`**.

Конкретизиране на типизирани класове

Преди да представим повече подробности за типизацията, нека погледнем как се използват типизираните класове. Използването на типизирани класове става по следния начин:

```
<class_name><concrete_type> <variable_name> =
new <class_name><concrete_type>();
```

Отново, подобно на заместителя **`T`** в декларацията на нашия клас, знаците '**<**' и '**>**', които ограждат конкретния клас **`concrete_type`**, са задължителни.

Ако искаме да създадем два приюта, един за кучета и един за котки, ще трябва да използваме следния код:

```
AnimalShelter<Dog> dogsShelter = new AnimalShelter<Dog>();
AnimalShelter<Cat> catsShelter = new AnimalShelter<Cat>();
```

По този начин сме сигурни, че приютът **`dogsShelter`** винаги ще съдържа обекти от тип **`Dog`**, а променливата **`catsShelter`** ще оперира винаги с обекти от тип **`Cat`**.

Използване на неизвестните типове в декларация на полета

Веднъж използвани по време на декларацията на класа, параметрите, които са използвани за указване на неизвестните типове са видими в цялото тяло на класа, следователно могат да се използват за деклариране на полета както всеки друг тип:

```
[<modifiers>] T <field_name>;
```

Както можем да се досетим, в нашия пример с приюта за бездомни животни, можем да използваме тази възможност на езика C#, за да декларираме типа на полето **`animalsList`**, в което съхраняваме референции към обектите на приютените животни, вместо с конкретния тип **`Dog`**, с параметъра **`T`**:

```
private T[] animalList;
```

За сега нека приемем, че когато създаваме обект от нашия клас, подавайки конкретен тип (например **`Dog`**), по време на изпълнение на програмата неизвестният тип **T** ще бъде заменен с въпросния тип. Ако сме избрали да създадем приют за кучета, можем да смятаме, че нашето поле е декларирано по следния начин:

```
private Dog[] animalList;
```

Съответно, когато искаме да инициализираме въпросното поле в конструктора на нашия клас, ще трябва да го направим по същия начин, както обикновено – създаваме масив, само че използвайки заместителя на неизвестния тип – **T**:

```
public AnimalShelter(int placesNumber)
{
   animalList = new T[placesNumber]; // Initialization
   usedPlaces = 0;
}
```

Използване на неизвестните типове в декларация на методи

Тъй като един неизвестен тип, използван в декларацията на един типизиран клас е видим от отварящата до затварящата скоба на тялото на класа, освен за декларация на полета, той може да бъде използван и в декларацията на методи, а именно:

- Като параметър в списъка от параметри на метода:

```
<return_type> MethodWithParamsOfT(T param)
```

- Като резултат от изпълнението на метода:

```
T MethodWithReturnTypeOfT(<params>)
```

Както вече се досещаме, използвайки нашия пример, можем да адаптираме методите **Shelter** и **Release**, съответно:

- Като метод с параметър от неизвестен тип **T**:

```
public void Shelter(T newAnimal)
{
   // Method's body goes here ...
}
```

- И метод, който връща резултат от неизвестен тип **T**:

```
public T Release(int i)
{
   // Method's body goes here ...
}
```

Както вече знаем, когато създадем обект от нашия клас приют и неизвестния тип го заменим с някой конкретен тип (например `Cat`), по време на изпълнение на програмата горните методи ще имат следния вид:

- Параметърът на метода **`Shelter`** ще бъде от тип **`Cat`**:

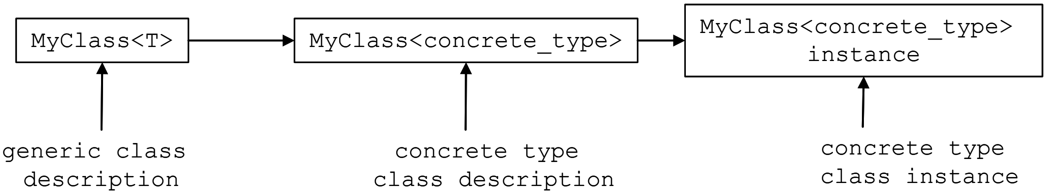
```
public void Shelter(Cat newAnimal)
{
   // Method's body goes here ...
}
```

- Методът **`Release`** ще връща резултат от тип **`Cat`**:

```
public Cat Release(int i)
{
   // Method's body goes here ...
}
```

Типизирането (generics) зад кулисите

Преди да продължим, нека обясним какво става в паметта на компютъра, когато работим с типизирани класове.


MyClass<T>
MyClass<concrete_type>
MyClass<concrete_type> instance
generic class description
concrete type class description
concrete type class instance

Първо декларираме нашия типизиран клас **`MyClass<T>`** (generic class description в горната схема). След това компилаторът транслира нашия код на междинен език (MSIL), като транслираният код, съдържа информация, че класът е типизиран, т.е. работи с неопределени до момента типове. По време на изпълнение, когато някой се опитва да работи с нашия типизиран клас и да го използва с конкретен тип, се създава ново **описание на клас** (concrete type class description в схемата по-горе), което е идентично с това на типизирания клас, с тази разлика, че навсякъде където е използвано **`T`**, сега се заменя с конкретния тип. Например ако се опитаме да използваме **`MyClass<int>`**, навсякъде, където в нашия код е използван неизвестния параметър **`T`**, ще бъде заменен с **`int`**. Едва след това, можем да създадем обект от типизирания клас с конкретен тип **`int`**. Особеното тук е, че за да се създаде този обект, ще се използва описанието на класа, което бе създадено междувременно (concrete type class description). Инстанцирането на шаблонен клас по дадени конкретни типове на неговите параметри се нарича "**специализация на тип**" или "**разгъване на шаблонен клас**".

Използвайки нашия пример, ако създадем обект от тип **AnimalShelter<T>**, който работи само с обекти от тип **Dog**, ако се опитаме да добавим обект от тип **Cat**, това ще доведе до грешка при компилация почти идентична с грешките, които бяха изведени при опит за добавяне на обект от тип **Cat**, към обект от тип **AnimalShelter**, който създадохме в първата подсекция "Приют за бездомни животни – пример":

```
public static void Main()
{
   AnimalShelter<Dog> dogsShelter = new AnimalShelter<Dog>(10);

   Cat cat1 = new Cat();

   dogsShelter.Shelter(cat1);
}
```

Както се очакваше, получаваме следните съобщения:

```
The best overloaded method match for 'AnimalShelter<
Dog>.Shelter(Dog)' has some invalid arguments

Argument 1: cannot convert from 'Cat' to 'Dog'
```

Типизиране на методи

Подобно на класовете, когато при декларацията на един метод, не можем да кажем от какъв тип ще са параметрите му, можем да типизираме метода. Съответно, указването на конкретния тип ще стане по време на извикване на метода, заменяйки непознатият тип с конкретен, както направихме при класовете.

Типизирането на метод се прави, като веднага след името и преди отварящата кръгла скоба на метода, се добави **<K>**, където **K** е заместителят на типа, който ще се използва в последствие:

```
<return_type> <methods_name><K>(<params>)
```

Съответно, можем да използваме неизвестния тип **K** за параметрите в списъка с параметри на метода **<params>**, чийто тип не ни е известен, а също и като връщана стойност или за деклариране на променливи от типа заместител **K** в тялото на метода.

Например, нека разгледаме един метод, който разменя стойностите на две променливи:

```
public void Swap<K>(ref K a, ref K b)
{
   K oldA = a;
   a = b;
   b = oldA;
}
```

Това е метод, който разменя стойностите на две променливи, без да се интересува от типа им. Затова сме го типизирали, за да можем да го прилагаме за всякакви типове променливи.

Съответно, ако искаме да разменим стойностите на две целочислени и след това на две низови променливи, бихме използвали нашия метод :

```
int num1 = 3;
int num2 = 5;
Console.WriteLine("Before swap: {0} {1}", num1, num2);
// Invoking the method with concrete type (int)
Swap<int>(ref num1, ref num2);
Console.WriteLine("After swap: {0} {1}\n", num1, num2);

string str1 = "Hello";
string str2 = "There";
Console.WriteLine("Before swap: {0} {1}!", str1, str2);
// Invoking the method with concrete type (string)
Swap<string>(ref str1, ref str2);
Console.WriteLine("After swap: {0} {1}!", str1, str2);
```

Когато изпълним този код, резултатът ще е както очакваме:

```
Before swap: 3 5
After swap: 5 3

Before swap: Hello There!
After swap: There Hello!
```

Забелязваме, че в списъка с параметри сме използвали също и ключовата дума **`ref`**. Това е така, заради спецификата на това което прави методът – а именно да размени стойностите на две референции. При използването на ключовата дума **`ref`**, методът ще използва същата референция, която е подадена от извикващия метод. По този начин, всички промени, които са направени от нашия метод върху тази променлива, ще се запазят след приключване работата на нашия метод и връщане на контрола върху изпълнението на програмата обратно на извикващия метод.

Трябва да знаем, че при извикване на типизиран метод, можем да пропуснем изричното деклариране на конкретния тип (в нашия пример **`<int>`**), тъй като компилаторът ще го установи автоматично, разпознавайки типа на подадените параметри. С други думи, нашият код може да бъде опростен използвайки следните извиквания:

```
Swap(ref num1, ref num2); // Invoking the method Swap<int>
Swap(ref str1, ref str2); // Invoking the method Swap<string>
```

Трябва да знаем, че компилаторът ще може да разпознае какъв е конкретния тип, само ако този тип участва в списъка с параметри. Компилаторът не може да разпознае какъв е конкретния тип на типизиран метод само от типа на връщаната стойност на метода или в случай, че методът е без параметри. В тези случаи, конкретния тип ще трябва да бъде подаден изрично. В нашия пример, това ще стане по подобие на първоначалното извикване на метода, чрез добавяне **`<int>`** или **`<string>`**.

Трябва да отбележим, че статичните методи също могат да бъдат типизирани за разлика от свойствата и конструкторите на класа.

Статичните методи също могат да бъдат типизирани, докато свойства и конструкторите на класа не могат.

Особености при деклариране на типизирани методи в типизирани класове

Както видяхме в секцията "Използване на неизвестните типове в декларацията на методи", нетипизираните методи могат да използват неизвестните типове, описани в декларацията на типизирания клас (например методите **`Shelter()`** и **`Release()`** от примера за приюта за бездомни животни):

AnimalShelter.cs

```
public class AnimalShelter<T>
{
   // ... rest of the code ...

   public void Shelter(T newAnimal)
   {
      // Method body here
   }

   public T Release(int i)
   {
      // Method body here
   }
}
```

Ако обаче, се опитаме да преизползваме променливата, с която сме означили непознатия тип на типизирания клас, например **`T`**, при декларацията на типизиран метод, тогава при опит за компилиране на класа, ще получим предупреждение (warning) CS0693, тъй като в областта на действие, на неизвестния тип **`T`**, дефиниран при декларацията на метода, припокрива областта на действие на неизвестния тип **`T`**, в декларацията на класа:

CommonOperations.cs

```
public class CommonOperations<T>
{
  // CS0693
  public void Swap<T>(ref T a, ref T b)
  {
    T oldA = a;
    a = b;
    b = oldA;
  }
}
```

При опит за компилация на този клас, ще получим следното съобщение:

```
Type parameter 'T' has the same name as the type parameter from
outer type 'CommonOperations<T>'
```

Затова, ако искаме нашият код да е гъвкав, и нашият типизиран метод безпроблемно да бъде извикван с конкретен тип, различен от този на типизирания клас при инстанцирането на класа, просто трябва да декларираме заместителя на неизвестния тип в декларацията на типизирания метод, да бъде различен от параметъра за неизвестния тип в декларацията на класа, както е показано по-долу:

CommonOperations.cs

```
public class CommonOperations<T>
{
  // No warning
  public void Swap<K>(ref K a, ref K b)
  {
    K oldA = a;
    a = b;
    b = oldA;
  }
}
```

По този начин, винаги ще сме сигурни, че няма да има препокриване на заместителите на неизвестните типове на метода и класа.

Използването на ключовата дума default в типизиран код

След като се запознахме с основите на типизирането, нека се опитаме да преработим нашия пръв пример в тази секция – класът описващ приют за бездомни животни. Както разбрахме, единственото, което е нужно да направим е да заменим конкретния тип **Dog** с някакъв параметър, например **T**:

AnimalsShelter.cs

```
public class AnimalShelter<T>
{
  private const int DefaultPlacesCount = 20;

  private T[] animalList;
  private int usedPlaces;

  public AnimalShelter()
    : this(DefaultPlacesCount)
  {
  }

  public AnimalShelter(int placesCount)
  {
    this.animalList = new T[placesCount];
    this.usedPlaces = 0;
  }

  public void Shelter(T newAnimal)
  {
    if (this.usedPlaces >= this.animalList.Length)
    {
      throw new InvalidOperationException("Shelter is full.");
    }
    this.animalList[this.usedPlaces] = newAnimal;
    this.usedPlaces++;
  }

  public T Release(int index)
  {
    if (index < 0 || index >= this.usedPlaces)
    {
      throw new ArgumentOutOfRangeException(
        "Invalid cell index: " + index);
    }
    T releasedAnimal = this.animalList[index];
    for (int i = index; i < this.usedPlaces - 1; i++)
    {
      this.animalList[i] = this.animalList[i + 1];
    }
    this.animalList[this.usedPlaces - 1] = null;
    this.usedPlaces--;

    return releasedAnimal;
  }
}
```

Всичко изглежда наред, докато не се опитаме да компилираме класа. Тогава получаваме следната грешка:

```
Cannot convert null to type parameter 'T' because it could be a
non-nullable value type. Consider using 'default(T)' instead.
```

Грешката е в метода **`Release()`** и е свързана със записването на резултат **`null`** в освободената последна (най-дясна) клетка на приюта. Проблемът е, че се опитваме да използваме подразбиращата се стойност за референтен тип, но не сме сигурни, дали конкретния тип е референтен или примитивен. Тъкмо затова, компилаторът извежда гореописаните грешки. Ако типът **`AnimalShelter`** се инстанцира по структура, а не по клас, то стойността null е невалидна.

За да се справим с този проблем, трябва в нашия код, вместо **`null`**, да използваме конструкцията **`default(T)`**, която връща подразбиращата се стойност за конкретния тип, който ще бъде използван на мястото на **`T`**. Както знаем подразбиращата стойност за референтен тип е **`null`**, а за числови типове – нула. Можем да направим следната промяна:

```
// this.animalList[this.usedPlaces - 1] = null;
this.animalList[this.usedPlaces - 1] = default(T);
```

Едва сега компилацията минава без проблем и класът **`AnimalShelter<T>`** работи коректно. Можем да го тестваме например по следния начин:

```
static void Main()
{
    AnimalShelter<Dog> shelter = new AnimalShelter<Dog>();
    shelter.Shelter(new Dog());
    shelter.Shelter(new Dog());
    shelter.Shelter(new Dog());
    Dog d = shelter.Release(1); // Release the second dog
    Console.WriteLine(d);
    d = shelter.Release(0); // Release the first dog
    Console.WriteLine(d);
    d = shelter.Release(0); // Release the third dog
    Console.WriteLine(d);
    d = shelter.Release(0); // Exception: invalid cell index
}
```

Предимства и недостатъци на типизирането

Типизирането на класове и методи води до по-голяма преизползваемост на кода, по-голяма сигурност и по-голяма ефективност, в сравнение с алтернативните нетипизирани решения.

Като генерално правило, програмистът трябва да се стреми към типизиране на класовете, които създава винаги, когато е възможно. Колкото повече се използва типизиране, толкова повече нивото на абстракция в програмата се покачва, както и самият код става по-гъвкав и преизползваем. Все пак трябва да имаме предвид, че прекалената употреба на типизиране може да доведе до прекалено генерализиране и кодът може да стане нечетим и труден за разбиране от други програмисти.

Ръководни принципи при именуването на заместителите при типизиране на класове и методи

Преди да приключим с темата за типизирането, нека дадем някои указания при работата със заместителите (параметрите) на непознатите типове в един типизиран клас:

1. Когато при типизирането имаме само един непознат тип, тогава е общоприето да се използва буквата **`T`**, като заместител за този непознат тип. Като пример можем да вземем декларацията на нашия клас **`AnimalShelter<T>`**, който използвахме до сега.
2. На заместителите трябва да се дават възможно най-описателните имена, освен ако една буква не е достатъчно описателна и добре подбрано име, не би подобрило по никакъв начин четимостта на кода. Например, можем да модифицираме нашия пример, заменяйки буквата **`T`**, с по-описателния заместител **`Animal`**:

AnimalShelter.cs

```
public class AnimalShelter<Animal>
{
   // ... rest of the code ...

   public void Shelter(Animal newAnimal)
   {
      // Method body here
   }

   public Animal Release(int i)
   {
      // Method body here
   }
}
```

Когато използваме описателни имена на заместителите, вместо буква, е добре да добавяме **`T`**, в началото на името, за да го разграничаваме по-лесно от имената на класовете в нашата програма. С други думи, вместо в предходния пример да използваме заместител **`Animal`**, е добре да използваме **`TAnimal`**.

Упражнения

1. Дефинирайте клас **`Student`**, който съдържа следната информация за студентите: трите имена, курс, специалност, университет, електронна поща и телефонен номер.

2. Декларирайте няколко конструктора за класа **`Student`**, които имат различни списъци с параметри (за цялостната информация за даден студент или част от нея). Данните, за които няма входна информация да се инициализират съответно с **`null`** или **`0`**.
3. Добавете статично поле в класа **`Student`**, в което се съхранява броя на създадените обекти от този клас.
4. Добавете метод в класа **`Student`**, който извежда пълна информация за студента.
5. Модифицирайте текущия код на класа **`Student`** така, че да капсулирате данните в класа чрез свойства.
6. Напишете клас **`StudentTest`**, който да тества функционалността на класа **`Student`**.
7. Добавете статичен метод в класа **`StudentTest`**, който създава няколко обекта от тип **`Student`** и ги съхранява в статични полета. Създайте статично свойство на класа, което да ги достъпва. Напишете тестова програма, която да извежда информацията за тях в конзолата.
8. Дефинирайте клас, който съдържа информация за мобилен телефон: модел, производител, цена, собственик, характеристики на батерията (модел, idle time и часове разговор /hours talk/) и характеристики на екрана (големина и цветове).
9. Декларирайте няколко конструктора за всеки от създадените класове от предходната задача, които имат различни списъци с параметри (за цялостната информация за даден студент или част от нея). Данните за полетата, които не са известни трябва да се инициализират съответно със стойности с **`null`** или **`0`**.
10. Към класа за мобилен телефон от предходните две задачи, добавете статично поле **`nokiaN95`**, което да съхранява информация за мобилен телефон модел Nokia 95. Добавете метод, в същия клас, който извежда информация за това статично поле.
11. Добавете изброим тип **`BatteryType`**, който съдържа стойности за тип на батерията (Li-Ion, NiMH, NiCd, …) и го използвайте като ново поле за класа **`Battery`**.
12. Добавете метод в класа **`GSM`**, който да връща информация за обекта под формата на **`string`**.
13. Дефинирайте свойства, за да капсулирате данните в класовете **`GSM`**, **`Battery`** и **`Display`**.
14. Напишете клас **`GSMTest`**, който тества функционалностите на класа **`GSM`**. Създайте няколко обекта от дадения клас и ги запазете в масив. Изведете информация за създадените обекти. Изведете информация за статичното поле **`nokiaN95`**.

15. Създайте клас **`Call`**, който съдържа информация за разговор, осъществен през мобилен телефон. Той трябва да съдържа информация за датата, времето на започване и продължителността на разговора.

16. Добавете свойство архив с обажданията – **`callHistory`**, което да пази списък от осъществените разговори.

17. В класа **`GSM`** добавете методи за добавяне и изтриване на обаждания (**`Call`**) в архива с обаждания на мобилния телефон. Добавете метод, който изтрива всички обаждания от архива.

18. В класа **`GSM`** добавете метод, който пресмята общата сума на обажданията (**`Call`**) от архива с обаждания на телефона (**`callHistory`**) като нека цената за едно обаждане се подава като параметър на метода.

19. Създайте клас **`GSMCallHistoryTest`**, с който да се тества функционалността на класа **`GSM`**, от задача 12, като обект от тип **`GSM`**. След това, към него добавете няколко обаждания (**`Call`**). Изведете информация за всяко едно от обажданията. Ако допуснем, че цената за минута разговор е 0.37, пресметнете и отпечатайте общата цена на разговорите. Премахнете най-дългият разговор от архива с обаждания и пресметнете общата цена за всички разговори отново. Най-накрая изтрийте архива с обаждания.

20. Нека е дадена библиотека с книги. Дефинирайте класове съответно за библиотека и книга. Библиотеката трябва да съдържа име и списък от книги. Книгите трябва да съдържат информация за заглавие, автор, издателство, година на издаване и ISBN-номер. В класа, който описва библиотека, добавете методи за добавяне на книга към библиотеката, търсене на книга по предварително зададен автор, извеждане на информация за дадена книга и изтриване на книга от библиотеката.

21. Напишете тестов клас, който създава обект от тип библиотека, добавя няколко книги към него и извежда информация за всяка една от тях. Имплементирайте тестова функционалност, която намира всички книги, чийто автор е Стивън Кинг и ги изтрива. Накрая, отново изведете информация за всяка една от оставащите книги.

22. Дадено ни е училище. В училището имаме класове и ученици. Всеки клас има множество от преподаватели. Всеки преподавател има множество от дисциплини, по които преподава. Учениците имат име и уникален номер в класа. Класовете имат уникален текстов идентификатор. Дисциплините имат име, брой уроци и брой упражнения. Задачата е да се моделира училище с C# класове. Трябва да декларирате класове заедно с техните полета, свойства, методи и конструктори. Дефинирайте и тестов клас, който демонстрира, че останалите класове работят коректно.

23. Напишете типизиран клас **`GenericList<T>`**, който пази списък от елементи от тип **`T`**. Пазете елементите от списъка в масив с фиксиран капацитет, който е зададен като параметър на конструктора на класа.

Добавете методи за добавяне на елемент, достъпване на елемент по индекс, премахване на елемент по индекс, вмъкване на елемент на зададена позиция, изчистване на списъка, търсене на елемент по стойност и предефинирайте метода **`ToString()`**.

24. Имплементирайте автоматично преоразмеряване на масива от предната задача, когато при добавяне на елемент се достигне капацитета на масива.
25. Дефинирайте клас **`Fraction`**, който съдържа информация за рационална дроб (например ¼, ½). Дефинирайте статичен метод **`Parse()`**, който да опитва да създаде дроб от символен низ (например -3/4). Дефинирайте подходящи свойства и конструктори на класа. Напишете и свойство от тип **`Decimal`**, което връща десетичната стойност на дробта (например 0.25).
26. Напишете клас **`FractionTest`**, който тества функционалността на класа от предната задача **`Fraction`**. Отделете специално внимание на тестването на функцията Parse с различни входни данни.
27. Напишете функция, която съкращава дробта (Например ако числителя и знаменателя са съответно 10 и 15, дробта да се съкращава до 2/3).

Решения и упътвания

1. Използвайте **`enum`** за специалностите и университетите.
2. За да избегнете повторение на код извиквайте конструкторите един от друг с **`this(<parameters>)`**.
3. Използвайте конструктора на класа като място, където броя на обектите от класа **`Student`** се увеличава.
4. Отпечатайте на конзолата всички полета от класа **`Student`**, следвани от празен ред.
5. Направете **`private`** всички членове на класа **`Student`**, след което използвайки Visual Studio (Refactor -> Encapsulate Field -> get and set accessor methods) дефинирайте автоматично публични методи за достъп до тези полета.
6. Създайте няколко студента и изведете цялата информация за всеки един от тях.
7. Можете да ползвате статичния конструктор, за да създадете инстанциите при първия достъп до класа.
8. Декларирайте три отделни класа: **`GSM`**, **`Battery`** и **`Display`**.
9. Дефинирайте описаните конструктори и за да проверите дали класовете работят правилно направете тестова програма.

10. Направете **private** полето и го инициализирайте в момента на декларацията му.
11. Използвайте **enum** за типа на батерията. Потърсете в интернет и други типове батерии на телефони, освен дадените в условието и добавете и тях като стойности на изброимия тип.
12. Предефинирайте метода **ToString()**.
13. В класовете **GSM**, **Battery** и **Display** дефинирайте подходящи **private** полета и генерирайте **get** / **set**. Можете да ползвате автоматичното генериране в Visual Studio.
14. Добавете метод **printInfo()** в класа **GSM**.
15. Прочетете за класа **List<T>** в Интернет. Класът **GSM** трябва да пази разговорите си в списък от тип **List<Call>**.
16. Връщайте като резултат списъка с разговорите.
17. Използвайте вградените методи на класа **List<T>**.
18. Понеже тарифата е фиксирана, лесно можете да изчислите сумарната цена на проведените разговори.
19. Следвайте директно инструкциите от условието на задачата.
20. Дефинирайте класове **Book** и **Library**. За списъка с книги ползвайте **List<Book>**.
21. Следвайте директно инструкциите от условието на задачата.
22. Създайте класове **School**, **SchoolClass**, **Student**, **Teacher**, **Discipline** и в тях дефинирайте съответните им полета, както са описани в условието на задачата. Не ползвайте за име на клас думата "**Class**", защото в C# тя има специално значение. Добавете методи за отпечатване на всички полета от всеки от класовете.
23. Използвайте знанията си за типизираните класове. Проверявайте всички входни параметри на методите, за да се подсигурите, че няма да достъпите елемент на невалидна позиция.
24. Когато се достигне капацитета на масива, създайте нов масив с двойно по-голям размер и копирайте старите елементи в новия.
25. Напишете клас с 2 **private decimal** полета, които пазят информация съответно за числителя и знаменателя на дробта. Направете подходящи свойства, които да капсулират информацията на дробта. Освен другите изисквания в задачата, предефинирайте по подходящ начин стандартните за всеки обект функции: **Equals**, **GetHashCode**, **ToString**.
26. Измислете подходящи тестове, на които вашата функция може да даде грешен резултат. Добра практика е първо да се пишат тестовете, а след тях конкретната реализация на функционалността.

27. Потърсете в интернет информация за "най-голям общ делител" и алгоритъм за пресмятането му. Разделете числителя и знаменателя на техния най-голям общ делител и ще получите съкратената дроб.

Telerik Academy

Присъедини се към Академията на Телерик!

АКАДЕМИЯТА НА ТЕЛЕРИК предоставя безплатно практическо обучение, насочено към всички млади хора, желаещи да станат умели .NET софтуерни инженери и да се присъединят към екипа на Телерик.

Академията подготвя бъдещите софтуерни специалисти за сложността, разнообразието и динамиката на днешната ИТ действителност, като за целта всички курсисти се обучават да работят с най-новите технологии на Майкрософт и черпят знания и опит от доказали се в областта на програмирането лектори и разработчици на софтуер.

В академията ще получите задълбочени знания и опит, изучавайки:

C#, .NET технологии, бази данни, SQL, Silverlight, WPF, ASP.NET, HTML5, разработка на мобилни приложения за iOS, Android и Windows Phone, основите на софтуерното инженерство

Академията на Телерик ви дава възможност да:

- **Учите напълно БЕЗПЛАТНО**
- **Изберете сред редица РАЗЛИЧНИ КУРСОВЕ**
- **Овладеете ОСНОВИТЕ на софтуерното инженерство**
- **Усвоите ПРОЦЕСА за разработка на софтуер**
- **Получите задълбочени теоретични и практически ИТ ПОЗНАНИЯ**
- **Станете умел .NET СОФТУЕРЕН ИНЖЕНЕР**
- **Започнете своята ИТ кариера в ТЕЛЕРИК – РАБОТОДАТЕЛ #1 в България за 2010 г.**

Само в рамките на две години **АКАДЕМИЯТА НА ТЕЛЕРИК** за софтуерни инженери успя да се наложи като безспорен лидер у нас в предлагането на допълнително обучение за софтуерни специалисти, спомагайки за успешния старт в кариерното развитие на стотици ентусиазирани младежи.

academy.telerik.com

academy@telerik.com   *facebook.com/TelerikAcademy*

telerik
deliver more than expected

Глава 15. Текстови файлове

В тази тема...

В настоящата тема ще се запознаем с основните принципи за работа с текстови файлове в C#. Ще разясним какво е това поток, за какво служи и как се ползва. Ще обясним какво е текстов файл и как се чете и пише в текстови файлове. Ще демонстрираме и обясним добрите практики за прихващане и обработка на изключения, възникващи при работата с файлове. Разбира се, всичко това ще онагледим и демонстрираме на практика с много примери.

Потоци

Потоците (streams) са важна част от всяка входно-изходна библиотека. Те намират своето приложение, когато програмата трябва да "прочете" или "запише" данни от или във външен източник на данни – файл, други компютри, сървъри и т.н. Важно е да уточним, че терминът **вход (input)** се асоциира с четенето на информация, а терминът **изход (output)** – със записването на информация.

Какво представляват потоците?

Потокът е наредена последователност от байтове, които се изпращат от едно приложение или входно устройство и се получават в друго приложение или изходно устройство. Тези байтове се изпращат и получават един след друг и винаги пристигат в същия ред, в който са били изпратени. Потоците са абстракция на комуникационен канал за данни, който свързва две устройства или програми.

Потоците са основното средство за обмяна на информация в компютърния свят. Чрез тях различни програми достъпват файловете на компютъра, чрез тях се осъществява и мрежова комуникация между отдалечени компютри. Много операции от компютърния свят могат да се разглеждат като четене или писане в поток. Например печатането на принтер е всъщност пращане на поредица байтове към поток, свързан със съответния порт, към който е свързан принтера. Възпроизвеждането на звук от звуковата карта може да стане като се изпратят някакви команди, следвани от семплирания звук (който представлява поредица от байтове). Сканирането на документи от скенер може да стане като на скенера се изпратят някакви команди (чрез изходен поток) и след това се прочете сканираното изображение (като входен поток). Така работата с почти всяко периферно устройство (видеокамера, фотоапарат, мишка, клавиатура, USB стик, звукова карта, принтер, скенер и други) може да се осъществи през абстракцията на потоците.

За да прочетем или запишем нещо от или във файл, трябва да отворим поток към дадения файл, да извършим четенето или писането и да затворим потока. Потоците могат да бъдат текстови или бинарни, но това разделение е свързано с интерпретацията на изпращаните и получаваните байтове. Понякога, за удобство серия байтове се разглеждат като текст (в предварително зададено кодиране) и това се нарича текстов поток.

Модерните сайтове в Интернет не могат без потоци и така наречения **streaming** (произлиза от stream, т.е. поток), който представлява поточно достъпване на обемни мултимедийни файлове, идващи от Интернет. Поточното аудио и видео позволява файловете да се възпроизвеждат преди цялостното им локално изтегляне, което прави съответния сайт по-интерактивен.

Основни неща, които трябва да знаем за потоците

Потоците се използват, за четене и запис на данни от и на различни устройства. Те улесняват комуникацията между програма и файл, програма и отдалечен компютър и т.н.

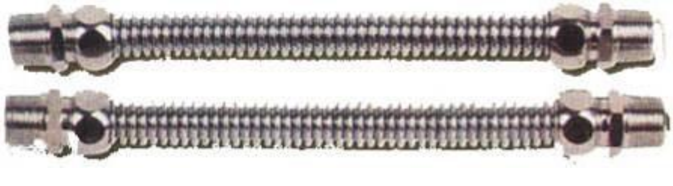
Потоците са **подредени** серии от байтове. Неслучайно наблягаме на думата подредени. От огромна важност е да се запомни, че потоците са строго подредени и организирани. По никакъв начин не можем да си позволим да влияем на подредбата на информацията в потока, защото по този начин ще я направим неизползваема. Ако един байт е изпратен към даден поток по-рано от друг, то той ще пристигне по-рано от него и това се гарантира от абстракцията "поток".

Потоците позволяват **последователен** достъп до данните си. Отново е важно да се вникне в значението на думата последователен. Може да манипулираме данните само в реда, в който те пристигат от потока. Това е тясно свързано с предходното свойство. Имайте това предвид, когато създавате собствени програми. Не можете да вземете първия байт, след това осмия, третия, тринадесетия и така нататък. Потоците **не** предоставят произволен достъп до данните си, а само **последователен**. Ако ви се струва по-лесно, може да мислим за потоците като за свързан списък от байтове, в който те имат строга последователност.

В различните ситуации се използват различни видове потоци. Едни потоци служат за работа с текстови файлове, други – за работа с бинарни (двоични) файлове, трети пък – за работа със символни низове. Различни са и потоците, които се използват при мрежова комуникация. Голямото изобилие от потоци ни улеснява в различните ситуации, но също така и ни затруднява, защото трябва да сме запознати със спецификата на всеки отделен тип, преди да го използваме в приложението си.

Потоците се отварят преди началото на работата с тях и се затварят след като е приключило използването им. Затварянето на потоците е нещо абсолютно задължително и не може да се пропусне, поради риск от загуба на данни, повреждане на файла, към който е отворен потока и т.н. – все неприятни неща, които не трябва да допускаме да се случват в нашите програми.

Потоците можем да оприличим на тръби, свързващи две точки:



От едната страна "наливаме" данни, а от другата данните "изтичат". Този, който налива данните, не се интересува как те се пренасят, но е сигурен, че каквото е налял, такова ще излезе от другата страна на тръбата. Тези, които ползват потоците, не се интересуват как данните стигат до тях. Те знаят, че ако някой налее нещо от другата страна, то ще пристигне при тях. Следователно можем да разглеждаме потоците са **транспортен канал за данни**, както и тръбите.

Основни операции с потоци

Когато работим с потоци в компютърните технологии, върху тях можем да извършваме следните операции:

Създаване

Свързваме потока с източник на данни, механизъм за пренос на данни или друг поток. Например, когато имаме файлов поток, тогава подаваме името на файла и режима, в който го отваряме (за четене, за писане или за четене и писане едновременно).

Четене

Извличаме данни от потока. Четенето винаги се извършва последователно от текущата позиция на потока. Четенето е блокираща операция и ако отсрещната страна не е изпратила данни докато се опитваме да четем или изпратените данни още не са пристигнали, може да се получи забавяне – от няколко милисекунди до часове, дни или по-голямо. Например, ако четем от мрежов поток, данните могат да се забавят по мрежата или отсрещната страна може изобщо да не изпрати никакви данни.

Запис

Изпращаме данни в потока по специфичен начин. Записът се извършва от текущата позиция на потока. Записът потенциално може да е блокираща операция и да се забави докато данните поемат по своя път. Например ако изпращаме обемни данните по мрежов поток, операцията може да се забави докато данните отпътуват по мрежата.

Позициониране

Преместване на текущата позиция на потока. Преместването се извършва спрямо текуща позиция, като можем да позиционираме спрямо текуща позиция, спрямо началото на потока, или спрямо края на потока. Преместване можем да извършваме единствено в потоци, които поддържат позициониране. Например файловите потоци обикновено поддържат позициониране, докато мрежовите не поддържат.

Затваряне

Приключваме работата с потока и освобождаваме ресурсите, заети от него. Затварянето трябва да се извършва възможно най-рано след приключване на работата с потока, защото ресурс, отворен от един потребител, обикновено не може да се ползва от останалите потребители (в това число от други програми на същия компютър, които се изпълняват паралелно на нашата програма).

Потоци в .NET – основни класове

В .NET Framework класовете за работа с потоци се намират в пространството от имена `System.IO`. Нека се концентрираме върху тяхната йерархия, организация и функционалност.

Можем да отличим два основни типа потоци – такива, които работят с двоични данни и такива, които работят с текстови данни. Ще се спрем на основните характеристики на тези два вида след малко.

На върха на йерархията на потоците стои абстрактен клас за входно-изходен поток. Той не може да бъде инстанциран, но дефинира основната функционалност, която притежават всички останали потоци.

Съществуват и буферирани потоци, които не добавят никаква допълнителна функционалност, но използват буфер при четене и записване на информацията, което значително повишава бързодействието. Буферираните потоци няма да се разглеждат в тази глава, тъй като ние се концентрираме върху обработката на текстови файлове. Ако имате желание, може да се допитате до богатата документация, достъпна в Интернет, или към някой учебник за по-напреднали в програмирането.

Някои потоци добавят допълнителна функционалност при четене и записване на данните. Например съществуват потоци, които компресират / декомпресират изпратените към тях данни и потоци, които шифрират и дешифрират данните. Тези потоци се свързват към друг поток (например файлов или мрежов поток) и добавят към неговата функционалност допълнителна обработка.

Основните класове в пространството от имена **`System.IO`** са **`Stream`** – базов абстрактен клас за всички потоци, **`BufferedStream`**, **`FileStream`**, **`MemoryStream`**, **`GZipStream`**, **`NetworkStream`**. Сега ще се спрем по-обстойно на някои от тях, разделяйки ги по основния им признак – типа данни, с който работят.

Всички потоци в C# си приличат и по едно основно нещо – задължително е да ги затворим, след като сме приключили работа с тях. В противен случай рискуваме да навредим на данните в потока или файла, към който сме го отворили. Това ни води и до първото основно правило, което винаги трябва да помним при работа с потоци:

> ⚠ **Винаги затваряйте потоците и файловете, с които работите! Оставянето на отворен поток или файл води до загуба на ресурси и може да блокира работата на други потребители или процеси във вашата система.**

Двоични и текстови потоци

Както споменахме по-рано, можем да разделим потоците на две големи групи в съответствие с типа данни, с който боравят, а именно – двоични потоци и текстови потоци.

Двоични потоци

Двоичните потоци, както личи от името им, работят с двоични (бинарни) данни. Сами се досещате, че това ги прави универсални и тях може да ползваме за четене на информация от всякакви файлове (картинки, музикални и мултимедийни файлове, текстови файлове и т.н.). Ще ги разгледаме съвсем накратко, защото за момента се ограничаваме до работа с текстови файлове.

Основните класове, които използваме, за да четем и пишем от и към двоични потоци са: **`FileStream`**, **`BinaryReader`** и **`BinaryWriter`**.

Класът **`FileStream`** ни предлага различни методи за четене и запис от бинарен файл (четене / запис на един байт и на поредица от байтове), пропускане на определен брой байтове, проверяване на броя достъпни байтове и, разбира се, метод за затваряне на потока. Обект от този клас може да получим, извиквайки конструктора му с параметър име на файл.

Класът **`BinaryWriter`** позволява записването в поток на данни от примитивни типове във вид на двоични стойности в специфично кодиране. Той има един основен метод – **`Write(…)`**, който позволява записване на всякакви примитивни типове данни – числа, символи, булеви стойности, масиви, стрингове и др. Класът **`BinaryReader`** позволява четенето на данни от примитивни типове, записани с помощта на **`BinaryWriter`**. Основните му методи ни позволяват да четем символ, масив от символи, цели числа, числа с плаваща запетая и др. Подобно на предходните два класа, обект от този клас може да получим, извиквайки конструктора му.

Текстови потоци

Текстовите потоци са много подобни на двоичните, но работят само с текстови данни или по-точно с поредици от символи (**`char`**) и стрингове (**`string`**). Идеални са за работа с текстови файлове. От друга страна това ги прави неизползваеми при работа с каквито и да е бинарни файлове.

Основните класове за работа с текстови потоци са **`TextReader`** и **`TextWriter`**. Те са абстрактни класове и от тях не могат да бъдат създавани обекти. Тези класове дефинират базова функционалност за четене и запис на класовете, които ги наследяват. По важните им методи са:

- **`ReadLine()`** – чете един текстов ред и връща символен низ.
- **`ReadToEnd()`** – чете всичко от потока до неговия край и връща символен низ.
- **`Write()`** – записва символен низ в потока.
- **`WriteLine()`** – записва един текстов ред в потока.

Както знаете, символите в .NET са Unicode символи, но потоците могат да работят освен с Unicode и с други кодирания (кодировки), например стандартното за кирилицата кодиране Windows-1251.

Класовете, на които ще обърнем най-голямо внимание в тази глава са `StreamReader` и `StreamWriter`. Те наследяват директно класовете `TextReader` и `TextWriter` и реализират функционалност за четене и запис на текстова информация от и във файл. За да създадем обект от `StreamReader` или `StreamWriter`, ни е нужен файл или символен низ с име и път до файла. Боравейки с тези класове, можем да използваме всички методи, с които вече сме добре запознати от работата ни с конзолата. Четенето и писането на конзолата приличат много на четенето и писането съответно от `StreamReader` и `StreamWriter`.

Връзка между текстовите и бинарните потоци

При писане на текст класът `StreamWriter` скрито от нас превръща текста в байтове преди да го запише на текущата позиция във файла. За целта той използва кодирането, което му е зададено по време на създаването му. По подобен начин работи и `StreamReader` класът. Той вътрешно използва `StringBuilder` и когато чете бинарни данни от файла, ги конвертира към текст преди да ги върне като резултат от четенето.

Запомнете, че в операционната система няма понятие "текстов файл". Файлът винаги е поредица от байтове, а дали е текстов или бинарен зависи от интерпретацията на тези байтове. Ако искаме да разглеждаме даден файл или поток като текстов, трябва да го четем и пишем с текстови потоци (`StreamReader` или `StreamWriter`), а ако искаме да го разглеждаме като бинарен (двоичен), трябва да го четем и пишем с бинарен поток (`FileStream`).

Трябва да обърнем внимание, че текстовите потоци работят с текстови редове, т.е. интерпретират бинарните данни като поредица от редове, разделени един от друг със символ за нов ред. **Символът за нов ред** не е един и същ за различните платформи и операционни системи. За UNIX и Linux той е `LF (0x0A)`, за Windows и DOS той е `CR + LF (0x0D + 0x0A)`, а за Mac OS (до версия 9) той е `CR (0x0A)`. Така четенето на един текстов ред от даден файл или поток означава на практика четене на поредица от байтове до прочитане на един от символите `CR` или `LF` и преобразуване на тези байтове до текст спрямо използваното в потока кодиране (encoding). Съответно писането на един текстов ред в текстов файл или поток означава на практика записване на бинарната репрезентация на текста (спрямо използваното кодиране), следвано от символа (или символите) за нов ред за текущата операционна система (например `CR + LF`).

Четене от текстов файл

Текстовите файлове предоставят идеалното решение за четене и записване на данни, които трябва да ползваме често, а са твърде обемисти, за да ги въвеждаме ръчно всеки път, когато стартираме програмата. Затова сега ще разгледаме как да четем и пишем текстови файлове с класовете от .NET Framework и езика C#.

Класът StreamReader за четене на текстов файл

C# предоставя множество начини за четене от файлове, но не всички са лесни и интуитивни за използване. Ето защо се спираме на **`StreamReader`** класа. Класът **`System.IO.StreamReader`** предоставя най-лесния начин за четене на текстов файл, тъй като наподобява четенето от конзолата, което до сега сигурно сте усвоили до съвършенство.

Четейки всичко до момента, е възможно да сте малко объркани. Вече обяснихме, че четенето и записването от и в текстови файлове става само и изключително с потоци, а същевременно **`StreamReader`** не се появи никъде в изброените по-горе потоци и не сте сигурни дали въобще е поток. Наистина, **`StreamReader`** не е поток, но може да работи с потоци. Той предоставя най-лесния и разбираем клас за четене от текстов файл.

Отваряне на текстов файл за четене

Може да създадем **`StreamReader`** просто по име на файл (или пълен път до файла), което значително ни улеснява и намалява възможностите за грешка. При създаването можем да уточним и кодирането (encoding). Ето пример как може да бъде създаден обект от класа **`StreamReader`**:

```
// Create a StreamReader connected to a file
StreamReader reader = new StreamReader("test.txt");

// Read file here...

// Close the reader resource after you've finished using it
reader.Close();
```

Първото, което трябва да направим, за да четем от текстов файл, е да създадем променлива от тип **`StreamReader`**, която да свържем с конкретен файл от файловата система на нашия компютър. За целта е нужно само да подадем като параметър в конструктора му името на желания файл. Имайте предвид, че ако файлът се намира в папката, където е компилиран проектът (поддиректория **`bin\Debug`**), то можем да подадем само конкретното му име. В противен случай може да подадем пълния път до файла или да използваме релативен път.

Редът код в горния пример, който създава обект от тип **`StreamReader`**, може да предизвика появата на грешка. Засега просто подавайте път до съществуващ файл, а след малко ще обърнем внимание и на обработката на грешки при работа с файлове.

Пълни и релативни пътища

При работата с файлове можем да използваме пълни пътища (например **`C:\Temp\example.txt`**) или релативни пътища, спрямо директорията, от която е стартирано приложението (примерно **`..\..\example.txt`**).

Ако използвате пълни пътища, при подаване на пълния път до даден файл не забравяйте да направите escaping на наклонените черти, които се използват за разделяне на папките. В C# това можете да направите по два начина – с двойна наклонена черта или с цитирани низове, започващи с `@` преди стринговия литерал. Например за да запишем в стринг пътя до файл "**`C:\Temp\work\test.txt`**" имаме два варианта:

```
string fileName = "C:\\Temp\\work\\test.txt";
string theSamefileName = @"C:\Temp\work\test.txt";
```

Въпреки, че използването на релативни пътища е по-трудно, тъй като трябва да съобразявате структурата на директориите на вашия проект, е силно препоръчително да избягвате пълните пътища.

Избягвайте пълни пътища и работете с относителни! Това прави приложението ви преносимо и по-лесно за инсталация и поддръжка.

Използването на пълен път до даден файл (примерно **`C:\Temp\test.txt`**) е лоша практика, защото прави програмата ви зависима от средата и непреносима. Ако я прехвърлите на друг компютър, ще трябва да коригирате пътищата до файловете, които тя търси, за да работи коректно. Ако използвате относителен (релативен) път спрямо текущата директория (например **`..\..\example.txt`**), вашата програма ще е лесно преносима.

Запомнете, че при стартиране на C# програма текущата директория е тази, в която се намира изпълнимият (`.exe`) файл. Най-често това е поддиректорията `bin\Debug` спрямо коренната директория на проекта. Следователно, за да отворите файла `example.txt` от коренната директория на вашия Visual Studio проект, трябва да използвате релативния път `..\..\example.txt`.

Отваряне на файл със задаване на кодиране

Както вече обяснихме, четенето и писането от и към текстови потоци изисква да се използва определено, предварително зададено кодиране на символите (character encoding). Кодирането може да се подаде при създаването на **`StreamReader`** обект като допълнителен втори параметър:

```
// Create a StreamReader connected to a file
StreamReader reader = new StreamReader("test.txt",
   Encoding.GetEncoding("Windows-1251"));

// Read file here...

// Close the reader resource after you've finished using it
reader.Close();
```

Като параметри в примера подаваме име на файла, който ще четем и обект от тип **`Encoding`**. Ако не бъде зададено специфично кодиране при отварянето на файла, се използва стандартното кодиране **`UTF-8`**. В показаният по-горе случай използваме кодиране **`Windows-1251`**. **`Windows-1251`** е 8-битов (еднобайтов) набор символи, проектиран от Майкрософт за езиците, използващи кирилица като български, руски и други. Кодиранията ще разгледаме малко по-късно в настоящата глава.

Четене на текстов файл ред по ред – пример

След като се научихме как да създаваме **`StreamReader`** вече можем да се опитаме да направим нещо по-сложно: да прочетем цял текстов файл ред по ред и да отпечатаме прочетеното на конзолата. Нашият съвет е да създавате текстовия файл в **`Debug`** папката на проекта (**`.\bin\Debug`**), така той ще е в същата директория, в която е вашето компилирано приложение и няма да се налага да подаваме пълния път до него при отварянето на файла. Нека нашият файл изглежда така:

sample.txt

```
This is our first line.
This is our second line.
This is our third line.
This is our fourth line.
This is our fifth line.
```

Имаме текстов файл, от който да четем. Сега трябва да създадем обект от тип **`StreamReader`** и да прочетем и отпечатаме всички редове. Това можем да направим по следния начин:

FileReader.cs

```
class FileReader
{
   static void Main()
   {
      // Create an instance of StreamReader to read from a file
      StreamReader reader = new StreamReader("Sample.txt");

      int lineNumber = 0;

      // Read first line from the text file
      string line = reader.ReadLine();

      // Read the other lines from the text file
      while (line != null)
      {
         lineNumber++;
         Console.WriteLine("Line {0}: {1}", lineNumber, line);
      line = reader.ReadLine();
    }

    // Close the resource after you've finished using it
    reader.Close();
  }
}
```

Сами се убеждавате, че няма нищо трудно в четенето на текстови файлове. Първата част на програмата вече ни е добре позната – създаваме променлива от тип **`StreamReader`** като в конструктора подаваме името на файла, от който ще четем. Параметърът на конструктора е пътят до файла, но тъй като нашият файл се намира в **`Debug`** директорията на проекта, ние задаваме като път само името му. Ако нашият файл се намираше в директорията на проекта, то тогава като път щяхме да подадем стринга - **`"..\..\Sample.txt"`**.

След това създаваме и една променлива – брояч, чиято цел е да брои и показва на кой ред от файла се намираме в текущия момент.

Създаваме и една променлива, която ще съхранява всеки прочетен ред. При създаването й направо четем първия ред от текстовия файл. Ако текстовият файл е празен, методът **`ReadLine()`** на обекта **`StreamReader`** ще върне **`null`**.

За същинската част – прочитането на файла ред по ред, използваме **`while`** цикъл. Условието за изпълнение на цикъла е докато в променливата **`line`** има записано нещо, т.е. докато има какво да четем от файла. В тялото на цикъла задачата ни се свежда до увеличаване на стойността на променливата-брояч с единица и след това да отпечатаме текущия ред от файла в желания от нас формат. Накрая отново с **`ReadLine()`** четем следващия ред от файла и го записваме в променливата **`line`**. За отпечатване използваме един метод, който ни е отлично познат от задачите, в които се е изисквало да се отпечата нещо на конзолата – **`WriteLine()`**.

След като сме прочели нужното ни от файла, отново не бива да забравяме да затворим обекта **`StreamReader`**, за да избегнем загубата на ресурси. За това ползваме метода **`Close()`**.

Винаги затваряйте инстанциите на `StreamReader` след като приключите работа с тях. В противен случай рискувате да загубите системни ресурси. За затваряне използвайте метода `Close()` или конструкцията `using`.

Резултатът от изпълнението на програмата би трябвало да изглежда така:

```
Line 1: This is our first line.
Line 2: This is our second line.
Line 3: This is our third line.
Line 4: This is our fourth line.
Line 5: This is our fifth line.
```

Автоматично затваряне на потока след приключване на работа с него

Както се забелязва в предния пример, след като приключихме работа с обекта от тип **`StreamReader`**, извикахме **`Close()`** и затворихме скрития поток, с който обектът **`StreamReader`** работи. Много често обаче начинаещите програмисти забравят да извикат **`Close()`** метода и с това излагат на опасност файла, от който четат, или в който записват. C# предлага конструкция за автоматично затваряне на потока или файла след приключване на работата с него. Тази конструкция е **`using`**. Синтаксисът й е следният:

```
using(<stream object>) { … }
```

Използването на **`using`** гарантира, че след излизане от тялото й автоматично ще се извика метода **`Close()`**. Това ще се случи дори ако при четенето на файла възникне някакво изключение.

След като вече знаем за **`using`** конструкцията, нека преработим предходния пример, така че да я използва:

FileReader.cs

```
class FileReader
{
   static void Main()
   {
      // Create an instance of StreamReader to read from a file
      StreamReader reader = new StreamReader("Sample.txt");

      using (reader)
      {
         int lineNumber = 0;

         // Read first line from the text file
         string line = reader.ReadLine();

         // Read the other lines from the text file
         while (line != null)
         {
            lineNumber++;
            Console.WriteLine("Line {0}: {1}", lineNumber, line);
            line = reader.ReadLine();
         }
```

```
        }
    }
}
```

Ако се чудите по какъв начин е най-добре да се грижите за затварянето на използваните във вашите програми потоци и файлове, следвайте следното правило:

Винаги използвайте `using` конструкцията в C# за да затваряте коректно отворените потоци и файлове!

Кодиране на файловете. Четене на кирилица

Нека сега разгледаме проблемите, които се появяват при четене с некоректно кодиране, например при четене на файл на кирилица.

Кодиране (encoding)

Добре знаете, че в паметта на компютрите всичко се запазва в двоичен вид. Това означава, че се налага и текстовите файлове да се представят цифрово, за да могат да бъдат съхранени в паметта, както и на твърдия диск. Този процес се нарича **кодиране на файловете**.

Кодирането се състои в заместването на текстовите символи (цифри, букви, препинателни знаци и т.н.) с точно определени поредици от числови стойности. Може да си го представите като голяма таблица, в която срещу всеки символ стои определена стойност (пореден номер).

Кодиращите схеми (character encodings) задават правила за преобразуване на текст в поредица от байтове и на обратно. Кодиращата схема е една таблица със символи и техните номера, но може да съдържа и специални правила. Например символът "ударение" (U+0300) е специален и се залепя за последния символ, който го предхожда. Той се кодира като един или няколко байта (в зависимост от кодиращата схема), но на него не му съответства никакъв символ, а част от символ. Ще разгледаме две кодирания, които се използват най-често при работа с кирилица: **`UTF-8`** и **`Windows-1251`**.

`UTF-8` е кодираща схема, при която най-често използваните символи (латинската азбука, цифри и някои специални знаци) се кодират в един байт, по-рядко използваните Unicode символи (като кирилица, гръцки и арабски) се кодират в два байта, а всички останали символи (китайски, японски и много други) се кодират в 3 или 4 байта. Кодирането UTF-8 може да преобразува произволен Unicode текст в бинарен вид и на обратно и поддържа всичките над 100 000 символа от Unicode стандарта. Кодирането UTF-8 е универсално и е подходящо за всякакви езици, азбуки и писмености.

Друго често използвано кодиране е **`Windows-1251`**, с което обикновено се кодират текстове на кирилица (например съобщения изпратени по e-mail).

То съдържа 256 символа, включващи латинската азбука, кирилицата и някои често използвани знаци. То използва по един байт за всеки символ, но за сметка на това някои символи не могат да бъдат записани в него (например символите от китайската азбука) и се губят при опит да се направи това. Това кодиране се използва по подразбиране в Windows, който е настроен за работа с български език.

Други примери за кодиращи схеми (encodings или charsets) са `ISO 8859-1`, `Windows-1252`, `UTF-16`, `KOI8-R` и т.н. Те се ползват в специфични региони по света и дефинират свои набори от символи и правила за преминаване от текст към бинарни данни и на обратно.

За представяне на кодиращите схеми в .NET Framework се използва класът `System.Text.Encoding`, който се създава по следния начин:

```
Encoding win1251 = Encoding.GetEncoding("Windows-1251"));
```

Четене на кирилица

Вероятно вече се досещате, че ако искаме да четем от файл, който съдържа символи от кирилицата, трябва да използваме правилното кодиране, което "разбира" тези специални символи. Обикновено в Windows среда текстовите файлове, съдържащи кирилица, са записани в кодиране `Windows-1251`. За да го използваме, трябва да го зададем като encoding на потока, който ще обработваме с нашия `StreamReader`.

Ако не укажем изрично кодиращата схема (encoding) за четене от файла, .NET Framework ще бъде използва по подразбиране encoding `UTF-8`.

Може би се чудите какво става, ако объркаме кодирането при четене или писане във файл. Възможни са няколко сценария:

- Ако ползваме само латиница, всичко ще работи нормално.
- Ако ползваме кирилица и четем с грешен encoding, ще прочетем т. нар. каракацили (познати още като джуджуфлечки или маймуняци). Това са безсмислени символи, които не могат да се прочетат.
- Ако записваме кирилица в кодиране, което не поддържа кирилската азбука (например `ASCII`), буквите от кирилицата ще бъдат заменени безвъзвратно със символа "`?`" (въпросителна).

При всички случаи това са неприятни проблеми, които може да не забележим веднага, а чак след време.

За да избегнете проблемите с неправилно кодирането на файловете, винаги задавайте кодирането изрично. Иначе програмата може да работи некоректно или да се счупи на по-късен етап.

Стандартът Unicode. Четене на Unicode

Unicode представлява индустриален стандарт, който позволява на компютри и други електронни устройства винаги да представят и манипулират по един и същи начин текст, написан на повечето от световните писмености. Той се състои от дефиниции на над 100 000 символа, както и разнообразни стандартни кодиращи схеми (encodings). Обединението на различните символи, което ни предлага Unicode, води до голямото му разпространение. Както знаете, символите в C# (типовете **`char`** и **`string`**) също се представят в Unicode.

За да прочетем символи, записани в Unicode, трябва да използваме някоя от поддържаните в този стандарт кодиращи схеми. Най-известен и широко използван е **`UTF-8`**. Можем да го зададем като кодираща схема по вече познатия ни начин:

```
StreamReader reader = new StreamReader("test.txt",
  Encoding.GetEncoding("UTF-8"));
```

Ако се чудите дали за четене на текстов файл на кирилица да ползвате кодиране **`Windows-1251`** или **`UTF-8`**, на този въпрос няма ясен отговор. И двата стандарта масово се ползват за записване на текстове на български език. И двете кодиращи схеми са позволени и може да ги срещнете.

Писане в текстов файл

Писането в текстови файлове е много удобен способ за съхранение на различни видове информация. Например, можем да записваме резултатите от изпълнението на дадена програма. Можем да ползваме текстови файлове, примерно направата на нещо като дневник (log) на програмата – удобен начин за следене кога се е стартирала, отбелязване на различни грешки при изпълнението и т.н.

Отново, както и при четенето на текстов файл, и при писането, ще използваме един подобен на конзолата клас, който се нарича **`StreamWriter`**.

Класът StreamWriter

Класът **`StreamWriter`** е част от пространството от имена **`System.IO`** и се използва изключително и само за работа с текстови данни. Той много наподобява класа **`StreamReader`**, но вместо методи за четене, предлага такива за записване на текст във файл. За разлика от другите потоци, преди да запише данните на желаното място, той ги превръща в байтове. **`StreamWriter`** ни дава възможност при създаването си да определим желания от нас encoding. Можем да създадем инстанция на класа по следния начин:

```
StreamWriter writer = new StreamWriter("test.txt");
```

В конструктора на класа можем да подадем като параметър както път до файл, така и вече създаден поток, в който ще записваме, а също и кодираща схема. Класът **StreamWriter** има няколко предефинирани конструктора, в зависимост от това дали ще пишем във файл или в поток. В примерите ще използваме конструктор с параметър път до файл. Пример за използването на конструктора на класа **StreamWriter** с повече от един параметър е следният:

```
StreamWriter writer = new StreamWriter("test.txt",
  false, Encoding.GetEncoding("Windows-1251"));
```

В този пример подаваме път до файл като първи параметър. Като втори подаваме булева променлива, която указва дали ако файлът вече съществува, данните да бъдат залепени на края на файла или файлът да бъде презаписан. Като трети параметър подаваме кодираща схема (encoding).

Примерните редове код отново може да предизвикат появата на грешка, но на обработката на грешки при работа с текстови файлове ще обърнем внимание малко по-късно в настоящата глава.

Отпечатване на числата от 1 до 20 в текстов файл – пример

След като вече можем да създаваме **StreamWriter**, ще го използваме по предназначение. Целта ни ще е да запишем в един текстов файл числата от 1 до 20, като всяко число е на отделен ред. Можем да го направим по следния начин:

```
class FileWriter
{
  static void Main()
  {
    // Create a StreamWriter instance
    StreamWriter writer = new StreamWriter("numbers.txt");

    // Ensure the writer will be closed when no longer used
    using(writer)
    {
      // Loop through the numbers from 1 to 20 and write them
      for (int i = 1; i <= 20; i++)
      {
        writer.WriteLine(i);
      }
    }
  }
}
```

Започваме като създаваме инстанция на **StreamWriter** по вече познатия ни от примера начин.

За да изведем числата от 1 до 20 използваме един `for`-цикъл. В тялото на цикъла използваме метода `WriteLine(…)`, който отново познаваме от работата ни с конзолата, за да запишем текущото число на нов ред във файла. Не бива да се притеснявате, ако файл с избраното от вас име не съществува. Ако случаят е такъв, той ще бъде автоматично създаден в папката на проекта, а ако вече съществува, ще бъде презаписан (ще бъде изтрито старото му съдържание). Резултатът има следния вид:

numbers.txt

```
1
2
3
…
20
```

За да сме сигурни, че след приключване на работата с файла той ще бъде затворен, използваме `using` конструкцията.

Не пропускайте да затворите потока след като приключите използването му! За затварянето му използвайте C# конструкцията `using`.

Когато искате да отпечатате текст на кирилица и се колебаете кое кодиране да ползвате, предпочитайте кодирането UTF-8. То е универсално и поддържа не само кирилица, но и всички широкоразпространени световни азбуки: гръцки, арабски, китайски, японски и т.н.

Обработка на грешки

Ако сте следили примерите до момента, сигурно сте забелязали, че при доста от операциите, свързани с файлове, могат да възникнат изключителни ситуации. Основните принципи и подходи за тяхното прихващане и обработка вече са ви познати от главата "Обработка на изключения". Сега ще се спрем малко на специфичните грешки при работа с файлове и най-добрите практики за тяхната обработка.

Прихващане на изключения при работа с файлове

Може би най-често срещаната грешка при работа с файлове е `FileNotFoundException` (от името и личи, че това изключение съобщава, че желаният файл не е намерен). Тя може да възникне при създаването на `StreamReader`.

Когато задаваме определен encoding при създаване на `StreamReader` или `StreamWriter`, може да възникне изключение `ArgumentException`. Това означава, че избраният от нас encoding не се поддържа.

Друга често срещана грешка е **`IOException`**. Това е базов клас за всички входно-изходни грешки при работа с потоци.

Стандартният подход при обработване на изключения при работа с файлове е следният: декларираме променливите от клас **`StreamReader`** или **`StreamWriter`** в **`try-catch`** блок. В блока ги инициализираме с нужните ни стойности и прихващаме и обработваме потенциалните грешки по подходящ начин. За затваряне на потоците използваме конструкция **`using`**. За да онагледим казаното до тук, ще дадем пример.

Прихващане на грешка при отваряне на файл – пример

Ето как можем да прихванем изключенията, настъпващи при работа с файлове:

```
class HandlingExceptions
{
   static void Main()
   {
      string fileName = @"somedir\somefile.txt";
      try
      {
         StreamReader reader = new StreamReader(fileName);
         Console.WriteLine(
            "File {0} successfully opened.", fileName);
         Console.WriteLine("File contents:");
         using (reader)
         {
            Console.WriteLine(reader.ReadToEnd());
         }
      }
      catch (FileNotFoundException)
      {
         Console.Error.WriteLine(
            "Can not find file {0}.", fileName);
      }
      catch (DirectoryNotFoundException)
      {
         Console.Error.WriteLine(
            "Invalid directory in the file path.");
      }
      catch (IOException)
      {
         Console.Error.WriteLine(
            "Can not open the file {0}", fileName);
      }
   }
}
```

Примерът демонстрира четене от файл и печатане на съдържанието му на конзолата. Ако случайно сме объркали името на файла или сме изтрили файла, ще бъде хвърлено изключение от тип **`FileNotFoundException`**. В **`catch`** блок прихващаме този тип изключение и ако евентуално такова възникне, ще го обработим по подходящ начин и ще отпечатаме на конзолата съобщение, че не може да бъде намерен такъв файл. Същото ще се случи и ако не съществува директория с името **`"somedir"`**. Накрая за подсигуряване сме добавили и **`catch`** блок за **`IOException`**. Там ще попадат всички останали входно-изходни изключения, които биха могли да настъпят при работата с файла.

Текстови файлове – още примери

Надяваме се теоретичните обяснения и примерите досега да са успели да ви помогнат да навлезете в тънкостите при работа с текстови файлове. Сега ще разгледаме още няколко по-комплексни примери с цел да затвърдим получените до момента знания и да онагледим как да ги ползваме при решаването на практически задачи.

Брой срещания на подниз във файл – пример

Ето как може да реализираме проста програма, която брои колко пъти се среща даден подниз в даден текстов файл. В примера нека търсим подниз "C#", а текстовият файл има следното съдържание:

sample.txt

```
This is our "Intro to Programming in C#" book.
In it you will learn the basics of C# programming.
You will find out how nice C# is.
```

Броенето можем да направим така: ще прочитаме файла ред по ред и всеки път, когато срещнем търсената от нас дума, ще увеличаваме стойността на една променлива (брояч). Ще обработим възможните изключителни ситуации, за да може потребителят да получава адекватна информация при появата на грешки. Ето и примерна реализация:

CountingWordOccurrences.cs

```
static void Main()
{
   string fileName = @"..\..\sample.txt";
   string word = "C#";
   try
   {
      StreamReader reader = new StreamReader(fileName);
      using (reader)
      {
            int occurrences = 0;
            string line = reader.ReadLine();
            while (line != null)
            {
                int index = line.IndexOf(word);
                while (index != -1)
                {
                    occurrences++;
                    index = line.IndexOf(word, (index + 1));
                }
                line = reader.ReadLine();
            }
            Console.WriteLine(
                "The word {0} occurs {1} times.", word, occurrences);
        }
    }
    catch (FileNotFoundException)
    {
        Console.Error.WriteLine(
            "Can not find file {0}.", fileName);
    }
    catch (IOException)
    {
        Console.Error.WriteLine(
            "Can not read the file {0}.", fileName);
    }
}
```

За краткост в примерния код думата, която търсим, е твърдо зададена (hardcoded). Вие може да реализирате програмата така, че да търси дума, въведена от потребителя.

Виждате, че примерът не се различава много от предишните. В него инициализираме променливите извън **`try-catch`** блока. Пак използваме **`while`**-цикъл, за да прочитаме редовете на текстовия файл един по един. Вътре в тялото му има още един **`while`**-цикъл, с който преброяваме колко пъти се среща думата в дадения ред и увеличаваме брояча на срещанията. Това става като използваме метода **`IndexOf(…)`** от класа **`String`** (припомнете си какво прави той в случай, че сте забравили). Не пропускаме да си гарантираме затварянето на **`StreamReader`** обекта използвайки **`using`** конструкцията. Единственото, което после ни остава да направим, е да изведем резултата в конзолата.

За нашия пример резултатът е следният:

```
The word C# occurs 3 times.
```

Коригиране на файл със субтитри – пример

Сега ще разгледаме един по-сложен пример, в който едновременно четем от един файл и записваме в друг. Става дума за програма, която коригира файл със субтитри за някакъв филм.

Нашата цел ще бъде да изчетем един файл със субтитри, които са некоректни и не се появяват в точния момент и да отместим времената по подходящ начин, за да се появяват правилно. Един такъв файл в общия случай съдържа времето за появяване на екрана, времето за скриване от екрана и текста, който трябва да се появи в дефинирания интервал от време. Ето как изглежда един типичен файл със субтитри:

GORA.sub

```
{1029}{1122}{Y:i}Капитане, системите са|в готовност.
{1123}{1270}{Y:i}Наляганeто е стабилно.|- Пригответе се за
кацане.
{1343}{1468}{Y:i}Моля, затегнете коланите|и се настанете по
местата си.
{1509}{1610}{Y:i}Координати 5.6|- Пет, пет, шест, точка ком.
{1632}{1718}{Y:i}Къде се дянаха|координатите?
{1756}{1820}Командир Логар,|всички говорят на английски.
{1821}{1938}Не може ли да преминем|на сръбски още от началото?
{1942}{1992}Може!
{3104}{3228}{Y:b}Г.О.Р.А.|филм за космоса
...
```

За да го коригираме, просто трябва да нанесем корекция във времето за показване на субтитрите. Такава корекция може да бъде отместване (добавяне или изваждане на някаква константа) или промяна на скоростта (умножаване по някакъв коефициент, примерно 1.05).

Ето и примерен код, с който може да реализираме такава програма:

FixingSubtitles.cs

```
using System;
using System.IO;

class FixingSubtitles
{
  const double COEFFICIENT = 1.05;
  const int ADDITION = 5000;
  const string INPUT_FILE = @"..\..\source.sub";
  const string OUTPUT_FILE = @"..\..\fixed.sub";

  static void Main()
  {
    try
    {
      // Getting the Cyrillic encoding
      System.Text.Encoding encoding =
        System.Text.Encoding.GetEncoding(1251);

      // Create reader with the Cyrillic encoding
      StreamReader streamReader =
        new StreamReader(INPUT_FILE, encoding);

      // Create writer with the Cyrillic encoding
      StreamWriter streamWriter =
        new StreamWriter(OUTPUT_FILE, false, encoding);

      using (streamReader)
      {
        using (streamWriter)
        {
          string line;
          while ((line = streamReader.ReadLine()) != null)
          {
            streamWriter.WriteLine(FixLine(line));
          }
        }
      }
    }
    catch (IOException exc)
    {
      Console.WriteLine("Error: {0}.", exc.Message);
    }
  }

  static string FixLine(string line)
  {
    // Find closing brace
    int bracketFromIndex = line.IndexOf('}');

    // Extract 'from' time
    string fromTime = line.Substring(1, bracketFromIndex - 1);

    // Calculate new 'from' time
    int newFromTime = (int) (Convert.ToInt32(fromTime) *
      COEFFICIENT + ADDITION);

    // Find the following closing brace
    int bracketToIndex = line.IndexOf('}',
      bracketFromIndex + 1);

    // Extract 'to' time
    string toTime = line.Substring(bracketFromIndex + 2,
      bracketToIndex - bracketFromIndex - 2);
      // Calculate new 'to' time
      int newToTime = (int) (Convert.ToInt32(toTime) *
        COEFFICIENT + ADDITION);

      // Create a new line using the new 'from' and 'to' times
      string fixedLine = "{" + newFromTime + "}" + "{" +
        newToTime + "}" + line.Substring(bracketToIndex + 1);

      return fixedLine;
    }
}
```

В примера създаваме **StreamReader** и **StreamWriter** и задаваме да използват encoding "**Windows-1251**", защото ще работим с файлове, съдържащи кирилица. Отново използваме вече познатия ни начин за четене на файл ред по ред. Различното този път е, че в тялото на цикъла записваме всеки ред във файла с вече коригирани субтитри, след като го поправим в метода **FixLine(string)** (този метод не е обект на нашата дискусия, тъй като може да бъде имплементиран по много и различни начини в зависимост какво точно искаме да коригираме). Тъй като използваме **using** блокове за двата файла, си гарантираме, че те задължително се затварят, дори ако при обработката възникне изключение (това може да случи например, ако някой от редовете във файла не е в очаквания формат).

Упражнения

1. Напишете програма, която чете от текстов файл и отпечатва нечетните му редове на конзолата.
2. Напишете програма, която съединява два текстови файла и записва резултата в трети файл.
3. Напишете програма, която прочита съдържанието на текстов файл и вмъква номерата на редовете в началото на всеки ред и след това записва обратно съдържанието на файла.
4. Напишете програма, която сравнява ред по ред два текстови файла с еднакъв брой редове и отпечатва броя съвпадащи и броя различни редове.
5. Напишете програма, която чете от файл квадратна матрица от цели числа и намира подматрицата с размери 2 х 2 с най-голяма сума и записва тази сума в отделен текстов файл. Първият ред на входния файл съдържа големината на записаната матрица (N). Следващите N реда съдържат по N числа, разделени с интервал.

   Примерен входен файл:

```
4
2 3 3 4
```

```
0 2 3 4
3 7 1 2
4 3 3 2
```

Примерен изход: 17.

6. Напишете програма, която чете списък от имена от текстов файл, подрежда ги по азбучен ред и ги запазва в друг файл. Имената са записани по едно на ред.
7. Напишете програма, която заменя всяко срещане на подниза "`start`" с "`finish`" в текстов файл. Можете ли да пренапишете програмата така, че да заменя само цели думи? Работи ли програмата за големи файлове (примерно 800 MB)?
8. Напишете предната програма така, че да заменя само целите думи (не части от думи).
9. Напишете програма, която изтрива от текстов файл всички нечетни редове.
10. Напишете програма, която извлича от XML файл всичкия текст без таговете. Примерен входен файл:

```
<?xml version="1.0"><student><name>Pesho</name>
<age>21</age><interests count="3"><interest>
Games</instrest><interest>C#</instrest><interest>
Java</instrest></interests></student>
```

Примерен резултат:

```
Pesho
21
Games
C#
Java
```

11. Напишете програма, която изтрива от текстов файл всички думи, които започват с "`test`". Думите съдържат само символите 0...9, a…z, A…Z,_.
12. Даден е текстов файл `words.txt`, съдържащ списък от думи, по една на ред. Напишете програма, която изтрива от файла `text.txt` всички думи, които се срещат в другия файл. Прихванете всички възможни изключения (Exceptions).
13. Напишете програма, която прочита списък от думи от файл, наречен `words.txt`, преброява колко пъти всяка от тези думи се среща в друг файл `text.txt` и записва резултата в трети файл – `result.txt`, като преди това ги сортира по броя срещания в намаляващ ред. Прихванете всички възможни изключения (Exceptions).

Решения и упътвания

1. Използвайте примерите, които разгледахме в настоящата глава. Използвайте **`using`** конструкцията за да гарантиране коректното затваряне на входния и резултатния поток.

2. Ще трябва първо да прочетете първия входен файл ред по ред и да го запишете в резултатния файл в режим презаписване (overwrite). След това трябва да отворите втория входен файл и да го запишете ред по ред в резултатния файл в режим добавяне (append). За да създадете **`StreamWriter`** в режим презаписване / добавяне използвайте подходящ конструктор (намерете го в MSDN).

   Алтернативен начин е да прочетете двата файла в **`string`** с **`ReadToEnd()`**, да ги съедините в паметта и да ги запишете в резултатния файл. Този подход, обаче няма да работи за големи файлове (от порядъка на няколко гигабайта).

3. Следвайте примерите от настоящата глава. Помислете как бихте се справили със задачата, ако размера на файла е огромен (например няколко GB).

4. Следвайте примерите от настоящата глава. Ще трябва да отворите двата файла за четене едновременно и в цикъл да ги четете ред по ред заедно. Ако срещнете край на файл (т.е. прочетете **`null`**), който не съвпада с край на другия файл, значи двата файла съдържат различен брой редове и трябва да изведете съобщение за грешка.

5. Прочетете първия ред от файла и създайте матрица с прочетения размер. След това четете останалите редове един по един и отделяйте числата. След това ги записвайте на съответния ред в матрицата. Накрая намерете с два вложени цикъла търсената подматрица.

6. Записвайте всяко прочетено име в списък (**`List<string>`**), след това го сортирайте по подходящ начин (потърсете информация за метода **`Sort()`**) и накрая го отпечатайте в резултатния файл.

7. Четете файла ред по ред и използвайте методите на класа **`String`**. Ако зареждате целия файл в паметта вместо да го четете ред по ред, ще има проблеми при зареждане на големи файлове.

8. За всяко срещане на 'start' ще проверявате дали това е цялата дума или само част от дума.

9. Работете по аналогия на примерите от настоящата глава.

10. Четете входния файл символ по символ. Когато срещнете "<", значи започва таг, а когато срещнете ">" значи тагът завършва. Всички символи, които срещате, които са извън таговете, изграждат текста, който трябва да се извлече. Можете да го натрупвате в **`StringBuilder`** и да го печатате, когато срещнете "<" или достигнете края на файла.

11. Четете файла ред по ред и заменяйте думите, които започват с "**`test`**" с празен низ. За целта използвайте **`Regex.Replace(…)`** с подходящ регулярен израз. Алтернативно можете да търсите в прочетения ред от файла подниз "**`test`**" и всеки път, когато го намерите да вземете всички съседни на него букви вляво и вдясно. Така намирате думата, в която низът "**`test`**" участва и можете да я изтриете, ако започва с "**`test`**".

12. Задачата е подобна на предходната. Можете да четете текста ред по ред и да заменяте в него всяка от дадените думи с празен низ. Тествайте дали вашата задача обработва правилно изключенията като симулирате възможни сценарии (например липса на файл, липса на права за четене и писане и т.н.)002E

13. Създайте хеш-таблица с ключ думите от **`words.txt`** и стойност броя срещания на всяка дума (**`Dictionary<string, int>`**). Първоначално запишете в хеш-таблицата, че всички думи се срещат по 0 пъти. След това четете ред по ред файла **`text.txt`** и разделяйте всеки ред на думи. Проверявайте дали всяка от получените при разделянето думи се среща в хеш-таблицата и ако е така прибавяйте 1 към броя на срещанията й. Накрая запишете всички думи и броя им срещания в масив от тип **`KeyValuePair<string, int>`**. Сортирайте масива подавайки подходяща функция за сравнение, например по средния начин:

```
Array.Sort<KeyValuePair<string, int>>(
  arr, (a, b) => a.Value.CompareTo(b.Value));
```

Глава 16. Линейни структури от данни

В тази тема...

Много често, за решаване на дадена задача се нуждаем да работим с последователности от елементи. Например, за да прочетем тази книга, трябва да прочетем последователно всяка една страница т.е. да обходим последователно всеки един от елементите на множеството от нейните страници. В зависимост от конкретната задача се налага да прилагаме различни операции върху тази съвкупност от данни. В настоящата тема ще се запознаем с някои от основните представяния на данните в програмирането. Ще видим как при определена задача една структура е по-ефективна и удобна от друга. Ще разгледаме структурите "списък", "стек" и "опашка" както и тяхното приложение. Също така подробно ще се запознаем и с някои от реализациите на тези структури.

Абстрактни структури от данни

Преди да започнем разглеждането на класовете в C#, имплементиращи някои от най-често използваните структури от данни (като списъци и речници), ще разгледаме понятията **структури от данни** и **абстрактни структури от данни**.

Какво е структура данни?

Много често, когато пишем програми ни се налага да работим с множество от обекти (данни). Понякога добавяме и премахваме елементи, друг път искаме да ги подредим или да обработваме данните по друг специфичен начин. Поради това са изработени различни начини за съхранение на данните в зависимост от задачата, като най-често между елементите съществува някаква наредба (например обект А е преди обект Б).

В този момент на помощ ни идват **структурите от данни** – множество от данни организирани на основата на логически и математически закони. Много често изборът на правилната структура прави програмата много по-ефективна – можем да спестим памет и време за изпълнение.

Какво е абстрактен тип данни?

Най-общо **абстрактният тип данни (АТД)** дава определена дефиниция (абстракция) на конкретната структура т.е. определя допустимите операции и свойства, без да се интересува от конкретната реализация. Това позволява един тип абстрактни данни да има различни реализации и респективно различна ефективност.

Основни структури от данни в програмирането

Могат ясно да се различат няколко групи структури:

- Линейни – към тях спадат списъците, стековете и опашките
- Дървовидни – различни типове дървета
- Речници – хеш-таблици
- Множества

В настоящата тема ще разгледаме линейните (списъчни) структури от данни, а в следващите няколко теми ще обърнем внимание и на по-сложните структури като дървета, графи, хеш-таблици и множества и ще обясним кога се използва и прилага всяка от тези структури.

Овладяването на основните от структури от данни в програмирането е от изключителна важност, тъй като без тях не може да се програмира ефективно. Те, заедно с алгоритмите, са в основата на програмирането и в следващите няколко глави ще се запознаем с тях.

Списъчни структури

Най-често срещаните и използвани са линейните (списъчни) структури. Те представляват абстракция на всякакви видове редици, последователности, поредици и други подобни от реалния свят.

Списък

Най-просто можем да си представим списъка като наредена последователност (редица) от елементи. Нека вземем за пример списък за покупки от магазин. В списъка можем да четем всеки един от елементите (покупките), както и да добавяме нови покупки в него. Можем също така и да задраскваме (изтрием) покупки или да ги разместваме.

Абстрактна структура данни "списък"

Нека сега дадем една по-строга дефиниция на структурата списък:

Списък е линейна структура от данни, която съдържа поредица от елементи. Списъкът има свойството дължина (брой елементи) и елементите му са наредени последователно.

Списъкът позволява добавяне на елементи на всяко едно място, премахването им и последователното им обхождането. Както споменахме по-горе, един АТД може да има няколко реализации. Пример за такъв АТД е интерфейсът **`System.Collections.IList`**.

Интерфейсите в C# изграждат една "рамка" за техните имплементации – класовете. Тази рамка представлява съвкупност от методи и свойства, които всеки клас, имплементиращ интерфейса, трябва да реализира. Типът данни "интерфейс" в C# ще дискутираме подробно в главата "Принципи на обектно-ориентираното програмиране".

Всеки АТД реално определя някакъв интерфейс. Нека разгледаме интерфейса **`System.Collections.IList`**. Основните методи, които той декларира, са:

- **`int Add(object)`** - добавя елемент в края на списъка
- **`void Insert(int, object)`** – добавя елемент на предварително избрана позиция в списъка
- **`void Clear()`** – изтрива всички елементи от списъка
- **`bool Contains(object)`** – проверява дали елементът се съдържа в списъка
- **`void Remove(object)`** – премахва съответния елемент
- **`void RemoveAt(int)`** – премахва елемента на дадена позиция
- **`int IndexOf(object)`** – връща позицията на елемента

- **`this[int]`** – индексатор, позволява достъп на елементите по подадена позиция

Нека видим няколко от основните реализации на АТД списък и обясним в какви ситуации се използва всяка от тях.

Статичен списък (реализация чрез масив)

Масивите изпълняват много от условията на АТД списък, но имат една съществена разлика – списъците позволяват добавяне на нови елементи, докато масивите имат фиксиран размер.

Въпреки това е възможна реализация на списък чрез масив, който автоматично увеличава размера си при нужда (по подобие на класа **`StringBuilder`**). Такъв списък се нарича **статичен**. Ето една имплементация на статичен списък, реализиран чрез разширяем масив:

```
public class CustomArrayList
{
    private object[] arr;

    private int count;

    /// <summary>
    /// Returns the actual list length
    /// </summary>
    public int Count
    {
        get
        {
            return count;
        }
    }

    private static readonly int INITIAL_CAPACITY = 4;

    /// <summary>
    /// Initializes the array-based list – allocate memory
    /// </summary>
    public CustomArrayList()
    {
        arr = new object[INITIAL_CAPACITY];
        count = 0;
    }
```

Първо си създаваме масива, в който ще пазим елементите, както и брояч за това колко елемента имаме в момента. След това добавяме и конструктора, като инициализираме нашия масив с някакъв начален капацитет, за да не се налага да го преоразмеряваме, когато добавим нов елемент. Нека разгледаме някои от типичните операции:

```
/// <summary>
/// Adds element to the list
/// </summary>
/// <param name="item">The element you want to add</param>
public void Add(object item)
{
  Insert(count, item);
}

/// <summary>
/// Inserts the specified element at a given
/// position in this list
/// </summary>
/// <param name="index">
/// Index, at which the specified element is to be inserted
/// </param>
/// <param name="item">Element to be inserted</param>
/// <exception cref="System.IndexOutOfRangeException">Index is
invalid</exception>
public void Insert(int index, object item)
{
  if (index > count || index < 0)
  {
    throw new IndexOutOfRangeException(
      "Invalid index: " + index);
  }
  object[] extendedArr = arr;
  if (count + 1 == arr.Length)
  {
    extendedArr = new object[arr.Length * 2];
  }
  Array.Copy(arr, extendedArr, index);
  count++;
  Array.Copy(arr, index, extendedArr, index + 1, count - index -
1);
  extendedArr[index] = item;
  arr = extendedArr;
}
```

Реализирахме операцията добавяне на нов елемент, както и вмъкване на нов елемент. Тъй като едната операция е частен случай на другата, методът за добавяне вика този за вмъкване. Ако масивът ни се напълни заделяме два пъти повече място и копираме елементите от стария в новия масив.

Реализираме операциите търсене на елемент, връщане на елемент по индекс, и проверка за това дали даден елемент се съдържа в списъка:

```
/// <summary>
/// Returns the index of the first occurrence
/// of the specified element in this list.
/// </summary>
/// <param name="item">The element you are searching</param>
/// <returns>
/// The index of given element or -1 is not found
/// </returns>
public int IndexOf(object item)
{
   for (int i = 0; i < arr.Length; i++)
   {
      if (item == arr[i])
      {
         return i;
      }
   }

   return -1;
}

/// <summary>
/// Clears the list
/// </summary>
public void Clear()
{
   arr = new object[INITIAL_CAPACITY];
   count = 0;
}

/// <summary>
/// Checks if an element exists
/// </summary>
/// <param name="item">The item to be checked</param>
/// <returns>If the item exists</returns>
public bool Contains(object item)
{
   int index = IndexOf(item);
   bool found = (index != -1);
   return found;
}

/// <summary>
/// Retrieves the element on the set index
/// </summary>
/// <param name="index">Index of the element</param>
/// <returns>The element on the current position</returns>
public object this[int index]
{
   get
   {
        if (index >= count || index < 0)
        {
            throw new ArgumentOutOfRangeException(
                "Invalid index: " + index);
        }
        return arr[index];
    }
    set
    {
        if (index >= count || index < 0)
        {
            throw new ArgumentOutOfRangeException(
                "Invalid index: " + index);
        }
        arr[index] = value;
    }
}
```

Добавяме и операции за изтриване на елементи:

```
/// <summary>
/// Removes the element at the specified index
/// </summary>
/// <param name="index">
/// The index, whose element you want to remove
/// </param>
/// <returns>The removed element</returns>
public object Remove(int index)
{
    if (index >= count || index < 0)
    {
        throw new ArgumentOutOfRangeException(
            "Invalid index: " + index);
    }

    object item = arr[index];
    Array.Copy(arr, index + 1, arr, index, count - index + 1);
    arr[count - 1] = null;
    count--;

    return item;
}

/// <summary>
/// Removes the specified item
/// </summary>
/// <param name="item">The item you want to remove</param>
/// <returns>Item index or -1 if item does not exists</returns>
public int Remove(object item)
```

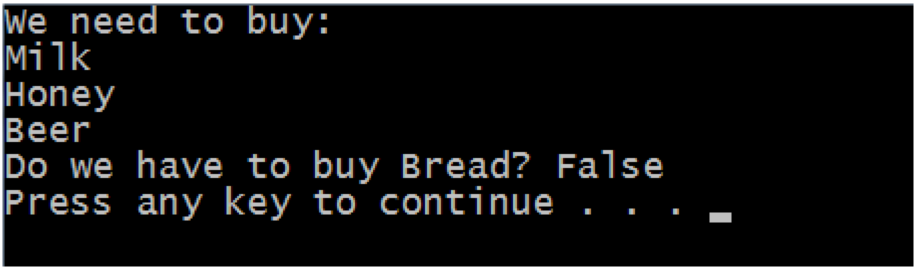
```
{
  int index = IndexOf(item);
  if (index == -1)
  {
    return index;
  }

  Array.Copy(arr, index + 1, arr, index, count - index + 1);
  count--;

  return index;
}
```

В горните методи премахваме елементи. За целта първо намираме търсения елемент, премахваме го, след което преместваме елементите след него, за да нямаме празно място на съответната позиция.

Нека сега разгледаме примерна употреба на класа, който току що създадохме. Добавен е и **`Main()`** метод, в който ще демонстрираме някои от операциите. В приложения код първо създаваме списък с покупки, а после го извеждаме на екрана. След това ще задраскаме маслините и ще проверим дали имаме да купуваме хляб.

```
public static void Main()
{
  CustomArrayList shoppingList = new CustomArrayList();
  shoppingList.Add("Milk");
  shoppingList.Add("Honey");
  shoppingList.Add("Olives");
  shoppingList.Add("Beer");
  shoppingList.Remove("Olives");
  Console.WriteLine("We need to buy:");
  for (int i = 0; i < shoppingList.Count; i++)
  {
    Console.WriteLine(shoppingList[i]);
  }
  Console.WriteLine("Do we have to buy Bread? " +
    shoppingList.Contains("Bread"));
}
```

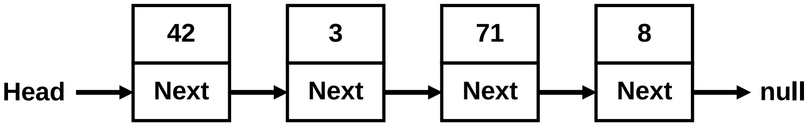
Ето как изглежда изходът от изпълнението на програмата:



```
We need to buy:
Milk
Honey
Beer
Do we have to buy Bread? False
Press any key to continue . . .
```

Свързан списък (динамична реализация)

Както видяхме, статичният списък има един сериозен недостатък – операциите добавяне и премахване от вътрешността на списъка изискват пренареждане на елементите. При често добавяне и премахване (особено при голям брой елементи) това може да доведе до ниска производителност. В такива случаи се използват т. нар. свързани списъци. Разликата при тях е в структурата на елементите – докато при статичния списък елементите съдържат само конкретния обект, при динамичния списък елементите пазят информация за следващия елемент. Ето как изглежда един примерен свързан списък в паметта:



За динамичната реализация ще са ни необходими два класа – класът **`Node`** – който ще представлява един отделен елемент от списъка и главният клас **`DynamicList`**:

```
/// <summary>
/// Represents dynamic list implementation
/// </summary>
public class DynamicList
{
  private class Node
  {
    private object element;
    private Node next;

    public object Element
    {
      get { return element; }
      set { element = value; }
    }

    public Node Next
    {
      get { return next; }
      set { next = value; }
    }

    public Node(object element, Node prevNode)
    {
      this.element = element;
      prevNode.next = this;
    }

    public Node(object element)
    {
        this.element = element;
        next = null;
      }
   }

   private Node head;
   private Node tail;
   private int count;
/*...*/
}
```

Нека разгледаме първо помощния клас **Node**. Той съдържа указател към следващия елемент, както и поле за обекта, който пази. Както виждаме класът е вътрешен за класа **DynamicList** (деклариран е в тялото на класа и е **private**) и следователно може да се достъпва само от него. За нашия **DynamicList** създаваме три полета **head** – указател към началния елемент, **tail** – указател към последния елемент и **count** – брояч за елементите.

След това декларираме и конструктор:

```
public DynamicList()
{
   this.head = null;
   this.tail = null;
   this.count = 0;
}
```

При първоначално конструиране списъкът е празен и затова **head = tail = null** и **count=0**.

Ще реализираме всички основни операции: добавяне и премахване на елементи, както и търсене на елемент.

Да започнем с операцията добавяне:

```
/// <summary>
/// Add element at the end of the list
/// </summary>
/// <param name="item">The element you want to add</param>
public void Add(object item)
{
   if (head == null)
   {
      // We have empty list
      head = new Node(item);
      tail = head;
   }
   else
   {
      // We have non-empty list
```

```
        Node newNode = new Node(item, tail);
        tail = newNode;
    }
    count++;
}
```

Разглеждат се два случая: празен списък и непразен списък. И в двата случая целта е да добавим елемента в края на списъка и след добавянето всички променливи (**`head`**, **`tail`** и **`count`** да имат коректни стойности).

Следва операцията изтриване по индекс. Тя е значително по-сложна от добавянето:

```
/// <summary>
/// Removes and returns element on the specific index
/// </summary>
/// <param name="index">
/// The index of the element you want to remove
/// </param>
/// <returns>The removed element</returns>
/// <exception cref="System.ArgumentOutOfRangeException">Index
is invalid</exception>
public object Remove(int index)
{
    if (index >= count || index < 0)
    {
        throw new ArgumentOutOfRangeException(
            "Invalid index: " + index);
    }

    // Find the element at the specified index
    int currentIndex = 0;
    Node currentNode = head;
    Node prevNode = null;
    while (currentIndex < index)
    {
        prevNode = currentNode;
        currentNode = currentNode.Next;
        currentIndex++;
    }

    // Remove the element
    count--;
    if (count == 0)
    {
        head = null;
    }
    else if (prevNode == null)
    {
        head = currentNode.Next;
```

```
   }
   else
   {
      prevNode.Next = currentNode.Next;
   }

   // Find last element
   Node lastElement = null;
   if (this.head != null)
   {
      lastElement = this.head;
      while (lastElement.Next != null)
      {
         lastElement = lastElement.Next;
      }
   }
   tail = lastElement;

   return currentNode.Element;
}
```

Първо се проверява дали посоченият за изтриване индекс съществува и ако не съществува се хвърля подходящо изключение. След това се намира елементът за изтриване чрез придвижване от началото на списъка към следващия елемент **`index`** на брой пъти. След като е намерен елементът за изтриване (**`currentNode`**), той се изтрива като се разглеждат 3 възможни случая:

- Списъкът остава празен след изтриването → изтриваме целия списък (**`head = null`**).
- Елементът е в началото на списъка (няма предходен) → правим **`head`** да сочи елемента веднага след изтрития (или в частност към **`null`**, ако няма такъв).
- Елементът е в средата или в края на списъка → насочваме елемент преди него да сочи към елемента след него (или в частност към **`null`**, ако няма следващ).

Накрая пренасочваме **`tail`** `към края на списъка.`

Следва реализацията на изтриването на елемент по стойност:

```
/// <summary>
/// Removes the specified item and return its index
/// </summary>
/// <param name="item">The item for removal</param>
/// <returns>The index of the element or -1 if does not
exist</returns>
public int Remove(object item)
{
    // Find the element containing searched item
    int currentIndex = 0;
    Node currentNode = head;
    Node prevNode = null;
    while (currentNode != null)
    {
        if ((currentNode.Element != null &&
             currentNode.Element.Equals(item)) ||
            (currentNode.Element == null) && (item == null))
        {
            break;
        }
        prevNode = currentNode;
        currentNode = currentNode.Next;
        currentIndex++;
    }

    if (currentNode != null)
    {
        // Element is found in the list. Remove it
        count--;
        if (count == 0)
        {
            head = null;
        }
        else if (prevNode == null)
        {
            head = currentNode.Next;
        }
        else
        {
            prevNode.Next = currentNode.Next;
        }

        // Find last element
        Node lastElement = null;
        if (this.head != null)
        {
            lastElement = this.head;
            while (lastElement.Next != null)
            {
                lastElement = lastElement.Next;
            }
        }
        tail = lastElement;

        return currentIndex;
    }
    else
    {
```

```
    // Element is not found in the list
    return -1;
  }
}
```

Изтриването по стойност на елемент работи като изтриването по индекс, но има 2 особености: търсеният елемент може и да не съществува и това налага допълнителна проверка; в списъка може да има елементи със стойност **`null`**, които трябва да предвидим и обработим по специален начин (вижте в кода). За да работи коректно изтриването, е необходимо елементите в масива да са сравними, т.е. да имат коректно реализирани методите **`Equals()`** ***и*** **`GetHashCode()`** от **`System.Object`**. Накрая отново намираме последния елемент и насочваме **`tail`** към него.

По-долу добавяме и операциите за търсене и проверка дали се съдържа даден елемент:

```
/// <summary>
/// Searches for given element in the list
/// </summary>
/// <param name="item">The item you are searching for</param>
/// <returns>
/// the index of the first occurrence of the element
/// in the list or -1 when not found
/// </returns>
public int IndexOf(object item)
{
  int index = 0;
  Node current = head;
  while (current != null)
  {
    if ((current.Element != null &&
       current.Element == item) ||
      (current.Element == null) && (item == null))
    {
      return index;
    }
    current = current.Next;
    index++;
  }
  return -1;
}

/// <summary>
/// Check if the specified element exists in the list
/// </summary>
/// <param name="item">The item you are searching for</param>
/// <returns>
/// True if the element exists or false otherwise
/// </returns>
```

```
public bool Contains(object item)
{
   int index = IndexOf(item);
   bool found = (index != -1);
   return found;
}
```

Търсенето на елемент работи, както в метода за изтриване: започва се от началото на списъка и се преравят последователно следващите един след друг елементи докато не се стигне до края на списъка.

Остана да реализираме още две операции – достъп до елемент по индекс (използвайки индексатор) и извличане броя елементи на списъка (използвайки свойство):

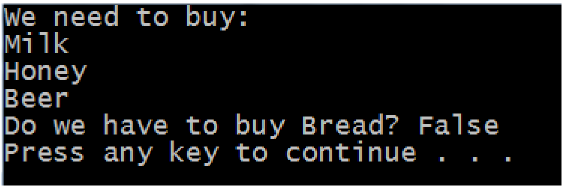
```
/// <summary>
/// Gets or sets the element on the specified position
/// </summary>
/// <param name="index">
/// The position of the element [0 … count-1]
/// </param>
/// <returns>The object at the specified index</returns>
/// <exception cref="System.ArgumentOutOfRangeException">
/// Index is invalid
/// </exception>
public object this[int index]
{
   get
   {
      if (index >= count || index < 0)
      {
         throw new ArgumentOutOfRangeException(
            "Invalid index: " + index);
      }
      Node currentNode = this.head;
      for (int i = 0; i < index; i++)
      {
         currentNode = currentNode.Next;
      }
      return currentNode.Element;
   }
   set
   {
      if (index >= count || index < 0)
      {
         throw new ArgumentOutOfRangeException(
            "Invalid index: " + index);
      }
      Node currentNode = this.head;
      for (int i = 0; i < index; i++)
```

```
        {
            currentNode = currentNode.Next;
        }
        currentNode.Element = value;

    }
}

/// <summary>
/// Gets the number of elements in the list
/// </summary>
public int Count
{
    get
    {
        return count;
    }
}
```

Нека видим накрая и нашия пример, този път реализиран чрез динамичен свързан списък.

```
public static void Main()
{
    DynamicList shoppingList = new DynamicList();
    shoppingList.Add("Milk");
    shoppingList.Add("Honey");
    shoppingList.Add("Olives");
    shoppingList.Add("Beer");
    shoppingList.Remove("Olives");
    Console.WriteLine("We need to buy:");
    for (int i = 0; i < shoppingList.Count; i++)
    {
        Console.WriteLine(shoppingList[i]);
    }
    Console.WriteLine("Do we have to buy Bread? " +
        shoppingList.Contains("Bread"));
}
```

Както и очакваме, резултатът е същият както при реализацията на списък чрез масив:

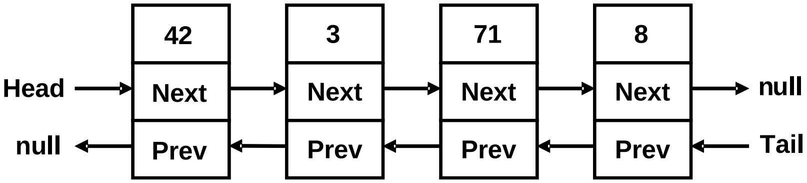


```
We need to buy:
Milk
Honey
Beer
Do we have to buy Bread? False
Press any key to continue . . .
```

Това показва, че можем да реализираме една и съща абстрактна структура от данни по фундаментално различни начини, но в крайна сметка ползвателите на структурата няма да забележат разлика в резултатите при използването ѝ. Разлика обаче има и тя е в скоростта на работа и в обема на заеманата памет.

Двойно свързани списъци

Съществува и т. нар. **двойно свързан списък** (двусвързан списък), при който всеки елемент съдържа стойността си и два указателя – към предходен и към следващ елемент (или **`null`**, ако няма такъв). Това ни позволява да обхождаме списъка, както напред така и назад. Това позволява някои операции да бъдат реализирани по ефективно. Ето как изглежда един примерен двусвързан списък в паметта:



Класът ArrayList

След като се запознахме с някои от основните реализации на списъците, ще се спрем на класовете в C#, които ни предоставят списъчни структури "на готово". Първият от тях е класът **`ArrayList`**, който представлява динамично-разширяем масив. Той е реализиран по сходен начин със статичната реализация на списък, която разгледахме по-горе. **`ArrayList`** дава възможност да добавяме, премахваме и търсим елементи в него. Някои по-важни членове, които можем да използваме са:

- **`Add(object)`** – добавяне на нов елемент
- **`Insert(int, object)`** – добавяне на елемент на определено място (индекс)
- **`Count`** – връща броя на елементите в списъка
- **`Remove(object)`** – премахване на определен елемент
- **`RemoveAt(int)`** – премахване на елемента на определено място (индекс)
- **`Clear()`** – изтрива елементите на списъка
- **`this[int]`** – индексатор, позволява достъп на елементите по подадена позиция

Както видяхме, един от основните проблеми при тази реализация е преоразмеряването на вътрешния масив при добавянето и премахването на елементи. В класа **`ArrayList`** проблемът е решен чрез предварително създаване на по-голям масив, който ни предоставя възможност да добавяме елементи, без да преоразмеряваме масива при всяко добавяне или премахване на елементи. След малко ще обясним това в детайли.

Класът ArrayList – пример

Класът **`ArrayList`** може да съхранява всякакви елементи – числа, символни низове и други обекти. Ето един малък пример:

```
using System;
using System.Collections;

class ProgrArrayListExample
{
  public static void Main()
  {
    ArrayList list = new ArrayList();
    list.Add("Hello");
    list.Add(5);
    list.Add(3.14159);
    list.Add(DateTime.Now);

    for (int i = 0; i < list.Count; i++)
    {
      object value = list[i];
      Console.WriteLine(
        "Index={0}; Value={1}\n", i, value);
    }
  }
}
```

В примера създаваме **`ArrayList`** и добавяме в него няколко елемента от различни типове: **`string`**, **`int`**, **`double`** и **`DateTime`**. След това итерираме по елементите и ги отпечатваме. Ако изпълним примера, ще получим следния резултат:

```
Index=0; Value=Hello
Index=1; Value=5
Index=2; Value=3.14159
Index=3; Value=29.12.2009 23:17:01
```

ArrayList с числа – пример

Ако искаме да си направим масив от числа и след това да обработим числата, примерно да намерим тяхната сума, се налага да преобразуваме типа **`object`** към число. Това е така, защото **`ArrayList`** всъщност е списък от обекти от тип **`object`**, а не от някой по-конкретен тип. Ето примерен код, който сумира елементите на **`ArrayList`**:

```
ArrayList list = new ArrayList();
list.Add(2);
list.Add(3);
list.Add(4);
int sum = 0;
for (int i = 0; i < list.Count; i++)
{
  int value = (int)list[i];
  sum = sum + value;
}
Console.WriteLine("Sum = " + sum);

// Output: Sum = 9
```

Преди да разгледаме още примери за работа с класа **`ArrayList`** ще да се запознаем с една концепция в C#, наречена "шаблонни типове данни". Тя дава възможност да се параметризират списъците и колекциите в C# и улеснява значително работата с тях.

Шаблонни класове (generics)

Когато използваме класа **`ArrayList`**, а и всички класове, имплементиращи интерфейса **`System.IList`**, се сблъскваме с проблема, който видяхме по-горе: когато добавяме нов елемент от даден клас ние го предаваме като обект от тип **`object`**. Когато по-късно търсим даден елемент, ние го получаваме като **`object`** и се налага да го превърнем в изходния тип. Не ни се гарантира, обаче, че всички елементи в списъка ще бъдат от един и същ тип. Освен това превръщането от един тип в друг отнема време, което забавя драстично изпълнението на програмата.

За справяне с описаните проблеми на помощ идват шаблонните класове. Те са създадени да работят с един или няколко типа, като при създаването си ние указваме какъв точно тип обекти ще съхраняваме в тях. Създаването на инстанция от даден шаблонен тип, примерно **`GenericType`**, става като в счупени скоби се зададе типа, от който трябва да бъдат елементите му:

```
GenericType<T> instance = new GenericType<T>();
```

Този тип **`T`** може да бъде всеки наследник на класа **`System.Object`**, примерно **`string`** или **`DateTime`**. Ето няколко примера:

```
List<int> intList = new List<int>();
List<bool> boolList = new List<bool>();
List<double> realNumbersList = new List<double>();
```

Нека сега разгледаме някои от шаблонните колекции в C#.

Класът List<T>

`List<T>` е шаблонният вариант на **`ArrayList`**. При инициализацията на обект от тип **`List<T>`** указваме типа на елементите, който ще съдържа списъка, т.е. заместваме означения с **`T`** тип с някой истински тип данни (например число или символен низ).

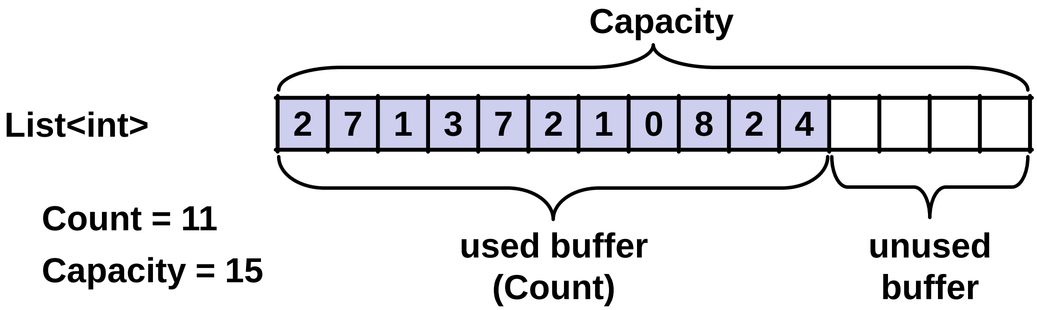
Нека разгледаме случай, в който искаме да създадем списък от целочислени елементи. Можем да го направим по следния начин:

```
List<int> intList = new List<int>();
```

Създаденият по този начин списък може да съдържа като стойности само цели числа и не може да съдържа други обекти, например символни низове. Ако се опитаме да добавим към **`List<int>`** обект от тип **`string`**, ще получим грешка по време на компилация. Чрез шаблонните типове компилаторът на C# ни пази от грешки при работа с колекции.

Класът List – представяне чрез масив

Класът **`List`** се представя в паметта като масив, от който една част съхранява елементите му, а останалите са свободни и се пазят като резервни. Благодарение на резервните празни елементи в масива операцията добавяне почти винаги успява да добави новия елемент без да разширява (преоразмерява) масива. Понякога, разбира се, се налага преоразмеряване, но понеже всяко преоразмеряване удвоява размера на масива, това се случва толкова рядко, че може да се пренебрегне на фона на броя добавяния. Можем да си представим един **`List`** като масив, който има някакъв капацитет и запълненост до определено ниво:



Благодарение на предварително заделеното пространство в масива, съхраняващ елементите на класа **`List<T>`**, той е изключително ефективна структура от данни, когато е необходимо бързо добавяне на елементи, извличане на всички елементи и пряк достъп до даден елемент по индекс.

Може да се каже, че **`List<T>`** съчетава добрите страни на списъците и масивите – бързо добавяне, променлив размер и директен достъп по индекс.

Кога да използваме List<T>?

Както вече обяснихме, класът **`List<T>`** използва вътрешно масив за съхранение на елементите, който удвоява размера си, когато се препълни. Тази негова специфика води до следните особености:

- Търсенето по индекс става много бързо – можем да достъпваме с еднаква скорост всеки един от елементите независимо от общия им брой.
- Търсенето по стойност на елемент работи с толкова сравнения, колкото са елементите (в най-лошия случай), т.е. не е бързо.
- Добавянето и премахването на елементи е бавна операция – когато добавяме или премахваме елементи, особено, ако те не се намират в края на списъка, се налага да разместваме всички останали елементи, а това е много бавна операция.
- При добавяне понякога се налага и увеличаване на капацитета на масива, което само по себе си е бавна операция, но се случва много рядко и средната скорост на добавяне на елемент към **`List`** не зависи от броя елементи, т.е. работи много бързо.

Използвайте List<T>, когато не очаквате често вмъкване и премахване на елементи, но очаквате да добавяте нови елементи в края или ползвате елементите по индекс.

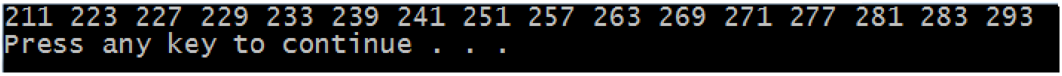
Прости числа в даден интервал – пример

След като се запознахме отвътре с реализацията на структурата списък и класа **`List<T>`**, нека видим как да използваме този клас. Ще разгледаме проблема за намиране на простите числа в някакъв интервал. За целта ще използваме следния алгоритъм:

```
public static List<int> GetPrimes(int start, int end)
{
    List<int> primesList = new List<int>();
    for (int num = start; num <= end; num++)
    {
        bool prime = true;
        double numSqrt = Math.Sqrt(num)
        for (int div = 2; div <= numSqrt; div++)
        {
            if (num % div == 0)
            {
                prime = false;
                break;
            }
        }
        if (prime)
        {
```

```
       primesList.Add(num);
     }
  }
  return primesList;
}

public static void Main()
{
  List<int> primes = GetPrimes(200, 300);
  foreach (var item in primes)
  {
     Console.WriteLine("{0} ", item);
  }
}
```

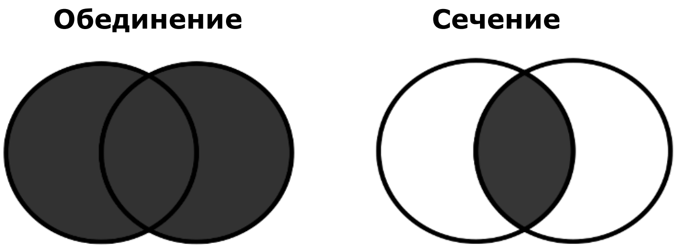
От математиката знаем, че ако едно число не е просто, то съществува поне един делител в интервала [2 … корен квадратен от даденото число]. Точно това използваме в примера по-горе. За всяко число търсим дали има делител в този интервал. Ако срещнем делител, то числото не е просто и можем да продължим със следващото. Постепенно чрез добавяне на прости числа пълним списъка, след което го обхождаме и го извеждаме на екрана. Ето го и изходът от горния код:



```
211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293
Press any key to continue . . .
```

Обединение и сечение на списъци – пример

Нека сега разгледаме един по-интересен пример - да напишем програма, която може да намира обединенията и сеченията на две множества числа.



Обединение **Сечение**

Можем да приемем, че имаме два списъка и искаме да вземем елементите, които се намират и в двата едновременно (сечение) или търсим тези, които се намират поне в единия от двата (обединение).

Нека разгледаме едно възможно решение на задачата:

```
public static List<int> Union(
     List<int> firstList, List<int> secondList)
{
  List<int> union = new List<int>();
  union.AddRange(firstList);
```

```
    foreach (var item in secondList)
    {
        if (!union.Contains(item))
        {
            union.Add(item);
        }
    }
    return union;
}

public static List<int> Intersect(List<int>
        firstList, List<int> secondList)
{
    List<int> intersect = new List<int>();
    foreach (var item in firstList)
    {
        if (secondList.Contains(item))
        {
            intersect.Add(item);
        }
    }

    return intersect;
}

public static void PrintList(List<int> list)
{
    Console.Write("{ ");
    foreach (var item in list)
    {
        Console.Write(item);
        Console.Write(" ");
    }
    Console.WriteLine("}");
}

public static void Main()
{
    List<int> firstList = new List<int>();
    firstList.Add(1);
    firstList.Add(2);
    firstList.Add(3);
    firstList.Add(4);
    firstList.Add(5);
    Console.Write("firstList = ");
    PrintList(firstList);

    List<int> secondList = new List<int>();
    secondList.Add(2);
    secondList.Add(4);
```

```
   secondList.Add(6);
   Console.Write("secondList = ");
   PrintList(secondList);

   List<int> unionList = Union(firstList, secondList);
   Console.Write("union = ");
   PrintList(unionList);

   List<int> intersectList =
      Intersect(firstList, secondList);
   Console.Write("intersect = ");
   PrintList(intersectList);
}
```

Програмната логика в това решение директно следва определенията за обединение и сечение на множества. Използваме операциите търсене на елемент в списък и добавяне на елемент към списък.

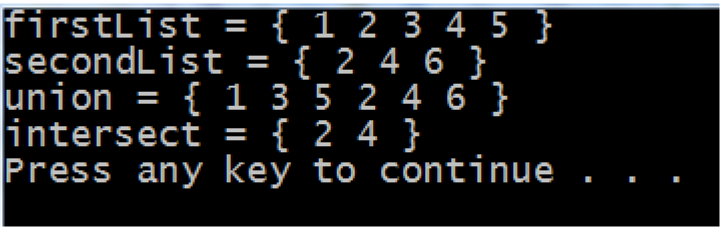
Ще решим проблема по още един начин: като използваме метода **`AddRange<T>(IEnumerable<T> collection)`** от класа **`List<T>`**:

```
public static void Main()
{
   List<int> firstList = new List<int>();
   firstList.Add(1);
   firstList.Add(2);
   firstList.Add(3);
   firstList.Add(4);
   firstList.Add(5);
   Console.Write("firstList = ");
   PrintList(firstList);

   List<int> secondList = new List<int>();
   secondList.Add(2);
   secondList.Add(4);
   secondList.Add(6);
   Console.Write("secondList = ");
   PrintList(secondList);

   List<int> unionList = new List<int>();
   unionList.AddRange(firstList);
   for (int i = unionList.Count-1; i >= 0; i--)
   {
      if (secondList.Contains(unionList[i]))
      {
         unionList.RemoveAt(i);
      }
   }
   unionList.AddRange(secondList);
   Console.Write("union = ");
```

```
  PrintList(unionList);

  List<int> intersectList = new List<int>();
  intersectList.AddRange(firstList);
  for (int i = intersectList.Count-1; i >= 0; i--)
  {
    if (!secondList.Contains(intersectList[i]))
    {
      intersectList.RemoveAt(i);
    }
  }
  Console.Write("intersect = ");
  PrintList(intersectList);
}
```

За да направим сечение правим следното: слагаме всички елементи от първия списък (чрез **`AddRange()`**), след което премахваме всички елементи, които не се съдържат във втория. Задачата може да бъде решена дори още по-лесно използвайки методът **`RemoveAll(Predicate<T> match)`**, но употребата му е обвързана с използване на конструкции наречени делегати и ламбда изрази, които се разглеждат в главата Ламбда изрази и LINQ заявки. Обединението правим като добавим елементите от първия списък, след което премахнем всички, които се съдържат във втория списък, след което добавяме всички елементи от втория списък.

Резултатът и от двете програми изглежда по един и същ начин:



```
firstList = { 1 2 3 4 5 }
secondList = { 2 4 6 }
union = { 1 3 5 2 4 6 }
intersect = { 2 4 }
Press any key to continue . . .
```

Превръщане на List в масив и обратното

В C# превръщането на списък в масив става лесно с използването на предоставения метод **`ToArray()`**. За обратната операция можем да използваме конструктора на **`List<T>(System.Array)`**. Нека видим пример демонстриращ употребата им:

```
public static void Main()
{
  int[] arr = new int[] { 1, 2, 3 };
  List<int> list = new List<int>(arr);
  int[] convertedArray = list.ToArray();
}
```

Класът LinkedList<T>

Този клас представлява динамична реализация на двусвързан списък. Елементите му пазят информация за обекта, който съхраняват, и указател към следващия и предишния елемент.

Кога да използваме LinkedList<T>?

Видяхме, че динамичната и статичните реализации имат специфика по отношение бързодействие на различните операции. С оглед на структурата на свързания списък трябва да имаме предвид следното:

- Можем да добавяме бързо на произволно място в списъка (за разлика от **`List<T>`**).
- Търсенето на елемент по индекс или по съдържание в **`LinkedList`** е бавна операция, тъй като се налага да обхождаме всички елементи последователно като започнем от началото на списъка.
- Изтриването на елемент е бавна операция, защото включва търсене.

Основни операции в класа LinkedList<T>

`LinkedList<T>` притежава същите операции като **`List<T>`**, което прави двата класа взаимнозаменяеми в зависимост от конкретната задача. По-късно ще видим, че **`LinkedList<T>`** се използва и при работа с опашки.

Кога да ползваме LinkedList<T>?

Класът **`LinkedList<T>`** е за предпочитане тогава, когато се налага добавяне/премахване на елементи на произволно място в списъка и когато достъпа до елементите е последователен. Когато обаче се търсят елементи или се достъпват по индекс, то **`List<T>`** се оказва по-подходящия избор. От гледна точка на памет, **`LinkedList<T>`** е по-икономичен, тъй като заделя памет за точно толкова елементи, колкото са текущо необходими.

Стек

Да си представим няколко кубчета, които сме наредили едно върху друго. Можем да слагаме ново кубче на върха, както и да махаме най-горното кубче. Или да си представим една ракла. За да извадим прибраните дрехи или завивки от дъното на раклата, трябва първо да махнем всичко, което е върху тях.

Точно тази конструкция представлява стекът – можем да добавяме елементи най-отгоре и да извличаме последния добавен елемент, но не и предходните (които са затрупани под него). Стекът е често срещана и използвана структура от данни. Стек се използва и вътрешно от C# виртуалната машина за съхранение на променливите в програмата и параметрите при извикване на метод.

Абстрактна структура данни "стек"

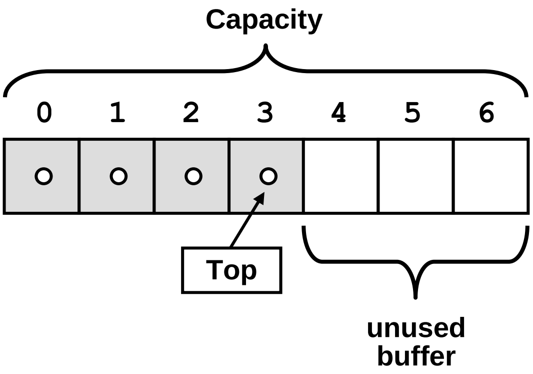
Стекът представлява структура от данни с поведение "последният влязъл първи излиза". Както видяхме в примера с кубчетата, елементите могат да се добавят и премахват само от върха на стека.

Структурата от данни стек също може да има различни реализации, но ние ще се спрем на двете основни – динамичната и статичната реализация.

Статичен стек (реализация с масив)

Както и при статичния списък можем да използваме масив за пазене на елементите на стека. Ще пазим индекс или указател, който сочи към елемента, който се намира на върха. Обикновено при запълване на масива следва заделяне на двойно повече памет, както това се случва при статичния списък (`ArrayList`).

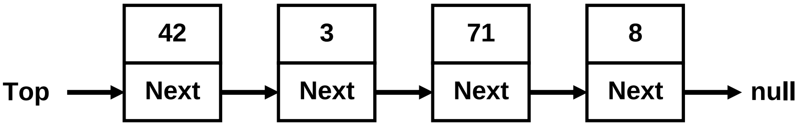
Ето как можем да си представим един статичен стек:



Както и при статичния масив се поддържа свободна буферна памет с цел по-бързо добавяне.

Свързан стек (динамична реализация)

За динамичната реализация ще използваме елементи, които пазят, освен обекта, и указател към елемента, който се намира "по-долу". Тази реализация решава ограниченията, които има статичната реализация както и необходимостта от разширяване на масива при нужда:



Когато стекът е празен, върхът има стойност `null`. При добавяне на нов елемент, той се добавя на мястото, където сочи върхът, след което върхът се насочва към новия елемент. Премахването става по аналогичен начин.

Класът Stack<T>

В C# можем да използваме имплементирания стандартно в .NET Framework клас **`System.Collections.Generics.Stack<T>`**. Той е имплементиран статично чрез масив, като масива се преоразмерява при необходимост.

Класът Stack<T> – основни операции

Реализирани са всички основни операции за работа със стек:

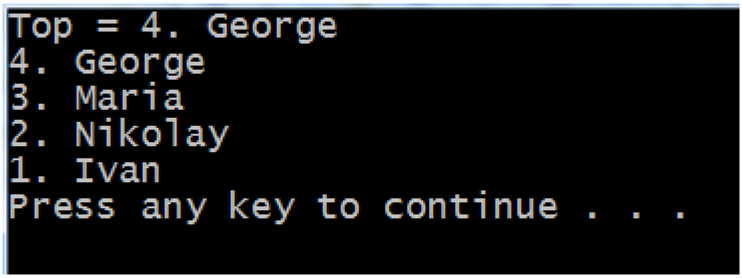
- **`Push(T)`** – добавя нов елемент на върха на стека
- **`Pop()`** – връща най-горния елемент като го премахва от стека
- **`Peek()`** – връща най горния елемент без да го премахва
- **`Count`** – връща броя на елементите в стека
- **`Clear()`** – премахва всички елементи
- **`Contains(T)`** – проверява дали елементът се съдържа в стека
- **`ToArray()`** – връща масив, съдържащ елементите от стека

Използване на стек – пример

Нека сега видим един прост пример как да използваме стек. Ще добавим няколко елемента, след което ще ги изведем на конзолата.

```
public static void Main()
{
  Stack<string> stack = new Stack<string>();
  stack.Push("1. Ivan");
  stack.Push("2. Nikolay");
  stack.Push("3. Maria");
  stack.Push("4. George");
  Console.WriteLine("Top = " + stack.Peek());
  while (stack.Count > 0)
  {
    string personName = stack.Pop();
    Console.WriteLine(personName);
  }
}
```

Тъй като стекът е структура "последен влязъл – пръв излязъл", програмата ще изведе записите в ред обратен на реда на добавянето. Ето какъв е нейният изход:



```
Top = 4. George
4. George
3. Maria
2. Nikolay
1. Ivan
Press any key to continue . . .
```

Проверка за съответстващи скоби – пример

Да разгледаме следната задача: имаме числов израз, на който искаме да проверим дали броят на отварящите скоби е равен на броя на затварящите. Спецификата на стека ни позволява да проверяваме дали скобата, която сме срещнали има съответстваща затваряща. Когато срещнем отваряща, я добавяме към стека. При срещане на затваряща вадим елемент от стека. Ако стекът остане празен преди края на програмата, в момент, в който трябва да извадим още един елемент, значи скобите са некоректно поставени. Същото важи и ако накрая в стека останат някакви елементи. Ето една примерна реализация:

```
public static void Main()
{
  string expression =
  "1 + (3 + 2 - (2+3) * 4 - ((3+1)*(4-2)))";
  Stack<int> stack = new Stack<int>();
  bool correctBrackets = true;

  for (int index = 0; index < expression.Length; index++)
  {
    char ch = expression[index];
    if (ch == '(')
    {
      stack.Push(index);
    }
    else if (ch == ')')
    {
      if (stack.Count == 0)
      {
        correctBrackets = false;
        break;
      }
      stack.Pop();
    }
  }
  if (stack.Count != 0)
  {
    correctBrackets = false;
  }
  Console.WriteLine("Are the brackets correct? " +
    correctBrackets);
}
```

Ето как изглежда изходът от примерната програма:

```
Are the brackets correct? True
```

Опашка

Структурата "**опашка**" е създадена да моделира опашки, като например опашка от чакащи документи за принтиране, чакащи процеси за достъп до общ ресурс и други. Такива опашки много удобно и естествено се моделират чрез структурата "опашка". В опашките можем да добавяме елементи само най-отзад и да извличаме елементи само най-отпред.

Нека, например, искаме да си купим билет за концерт. Ако отидем по-рано ще си купим първи от билетите. Ако обаче се забавим ще трябва да се наредим на опашката и да изчакаме всички желаещи преди нас да си купят билети. Това поведение е аналогично за обектите в АТД опашка.

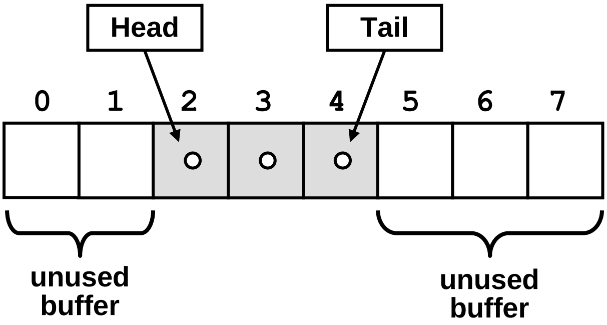
Абстрактна структура данни "опашка"

Абстрактната структура опашка изпълнява условието "първият влязъл първи излиза". Добавените елементи се нареждат в края на опашката, а при извличане поредният елемент се взима от началото (главата) ѝ.

Както и при списъка за структурата от данни опашка отново е възможна статична и динамична реализация.

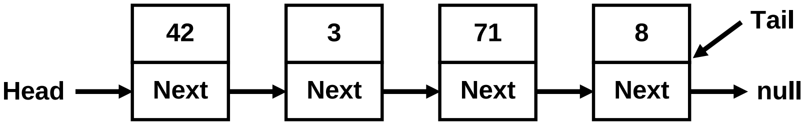
Статична опашка (реализация с масив)

В статичната опашка отново ще използваме масив за пазене на данните. При добавяне на елемент той се добавя на индекса, който следва края, след което края започва да сочи към ново добавения елемент. При премахване на елемент се взима елементът, към който сочи главата, след което главата започва да сочи към следващия елемент. По този начин опашката се придвижва към края на масива. Когато стигне до края, при добавяне на нов елемент той се добавя на първо място. Ето защо тази имплементация се нарича още **зациклена опашка**, тъй като мислено залепяме началото и края на масива и опашката обикаля в него:



Свързана опашка (динамична реализация)

Динамичната реализация на опашката много прилича на тази на свързания списък. Елементите отново съдържат две части – обекта и указател към предишния елемент:



Тук обаче елементите се добавят в края на опашката, а се вземат от главата, като нямаме право да взимаме или добавяме елементи на друго място.

Класът Queue<T>

В C# се използва статичната реализация на опашка чрез класа **`Queue<T>`**. Тук отново можем да укажем типа на елементите, с които ще работим, тъй като опашката и свързаният списък са шаблонни типове.

Класът Queue<T> – основни операции

`Queue<T>` ни предоставя основните операции характерни за структурата опашка. Ето някои от често използваните:

- **`Enqueue(T)`** – добавя елемент накрая на опашката
- **`Dequeue()`** – взима елемента от началото на опашката и го премахва
- **`Peek()`** – връща елементът от началото на опашката без да го премахва
- **`Clear()`** – премахва всички елементи от опашката
- **`Contains(T)`** – проверява дали елемента се съдържа в опашката

Използване на опашка – пример

Нека сега разгледаме прост пример. Да си създадем една опашка и добавим в нея няколко елемента. След това ще извлечем всички чакащи елементи и ще ги изведем на конзолата:

```
public static void Main()
{
   Queue<string> queue = new Queue<string>();
   queue.Enqueue("Message One");
   queue.Enqueue("Message Two");
   queue.Enqueue("Message Three");
   queue.Enqueue("Message Four");

   while (queue.Count > 0)
   {
      string msg = queue.Dequeue();
      Console.WriteLine(msg);
   }
}
```

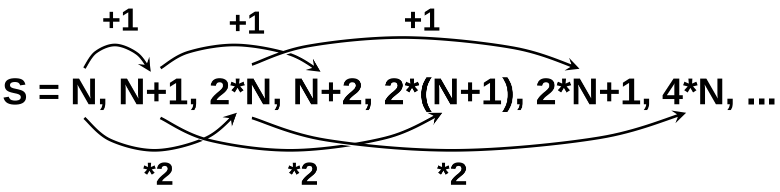
Ето как изглежда изходът на примерната програма:

```
Message One
Message Two
Message Three
Message Four
```

Вижда се, че елементите излизат от опашката в реда, в който са постъпили в нея.

Редицата N, N+1, 2*N – пример

Нека сега разгледаме задача, в която използването на структурата опашка ще бъде много полезна за реализацията. Да вземем редицата числа, чиито членове се получават по-следния начин: първият елемент е N; вторият получаваме като съберем N с 1; третият – като умножим първия с 2 и така последователно умножаваме всеки елемент с 2 и го добавяме накрая на редицата, след което го събираме с 1 и отново го поставяме накрая на редицата. Можем да илюстрираме този процес със следната фигура:



Както виждаме, процесът се състои във взимане на елементи от началото на опашка и поставянето на други в края ѝ. Нека сега видим примерна реализация, в която N=3 и търсим номера на член със стойност 16:

```
public static void Main()
{
   int n = 3;
```

```
    int p = 16;

    Queue<int> queue = new Queue<int>();
    queue.Enqueue(n);
    int index = 0;
    Console.WriteLine("S =");
    while (queue.Count > 0)
    {
        index++;
        int current = queue.Dequeue();
        Console.WriteLine(" " + current);
        if (current == p)
        {
            Console.WriteLine();
            Console.WriteLine("Index = " + index);
            return;
        }
        queue.Enqueue(current + 1);
        queue.Enqueue(2 * current);
    }
}
```

Ето как изглежда изходът е примерната програма:

```
S = 3 4 6 5 8 7 12 6 10 9 16
Index = 11
```

Както видяхме, стекът и опашката са две специфични структури с определени правила за достъпа до елементите в тях. Опашка използваме, когато очакваме да получим елементите в реда, в който сме ги поставили, а стек – когато елементите ни трябват в обратен ред.

Упражнения

1. Напишете програма, която прочита от конзолата поредица от цели положителни числа. Поредицата спира когато се въведе празен ред. Програмата трябва да изчислява сумата и средното аритметично на поредицата. Използвайте **`List<int>`**.
2. Напишете програма, която прочита N цели числа от конзолата и ги отпечатва в обратен ред. Използвайте класа **`Stack<int>`**.
3. Напишете програма, която прочита от конзолата поредица от цели положителни числа, поредицата спира когато се въведе празен ред, и ги сортира възходящо.
4. Напишете метод, който намира най-дългата подредица от равни числа в даден **`List<int>`** и връща като резултат нов **`List<int>`** със тази подредица. Напишете програма, която проверява дали този метод работи коректно.

5. Напишете програма, която премахва всички отрицателни числа от дадена редица.

   Пример: array = {19, -10, 12, -6, -3, 34, -2, 5} → {19, 12, 34, 5}

6. Напишете програма, която при дадена редица изтрива всички числа, които се срещат нечетен брой пъти.

   Пример: array = {4, 2, 2, 5, 2, 3, 2, 3, 1, 5, 2} → {5, 3, 3, 5}

7. Напишете програма, която по даден масив от цели числа в интервала [0..1000], намира по колко пъти се среща всяко число.

   Пример: array = {3, 4, 4, 2, 3, 3, 4, 3, 2}

   2 → 2 пъти

   3 → 4 пъти

   4 → 3 пъти

8. Мажорант на масив от N елемента е стойност, която се среща поне N/2+1 пъти. Напишете програма, която по даден масив от числа намира мажоранта на масива и го отпечатва. Ако мажоранта не съществува – отпечатва "The majorant does not exists!”.

   Пример: {2, 2, 3, 3, 2, 3, 4, 3, 3} → 3

9. Дадена е следната поредица:

   S1 = N;

   S2 = S1 + 1;

   S3 = 2*S1 + 1;

   S4 = S1 + 2;

   S5 = S2 + 1;

   S6 = 2*S2 + 1;

   S7 = S2 + 2;

   ...

   Използвайки класа **`Queue<T>`** напишете програма, която по дадено N отпечатва на конзолата първите 50 числа от тази поредица.

   Пример: N=2 → 2, 3, 5, 4, 4, 7, 5, 6, 11, 7, 5, 9, 6, ...

10. Дадени са числа N и M и следните операции:

    N = N+1

    N = N+2

    N = N*2

Напишете програма, която намира най-кратката поредица от посочените операции, която започва от N и завършва в M. Използвайте опашка.

Пример: N = 5, M = 16

Поредицата е: 5 → 7 → 8 → 16

11. Реализирайте структурата двойно свързан динамичен списък – списък, чиито елементи имат указател, както към следващия така и към предхождащия го елемент. Реализирайте операциите добавяне, премахване и търсене на елемент, добавяне на елемент на определено място (индекс), извличане на елемент по индекс и метод, който връща масив с елементите на списъка.

12. Създайте клас **`DynamicStack`** представляващ динамична реализация на стек. Добавете методи за необходимите операции.

13. Реализирайте структурата от данни "дек". Това е специфична списъчна структура, подобна на стек и опашка, позволяваща елементи да бъдат добавяни и премахвани от двата й края. Нека освен това, елемент поставен от едната страна да може да бъде премахнат само от същата. Реализирайте операции за премахване добавяне и изчистване на дека. При невалидна операция подавайте подходящо изключение.

14. Реализирайте структурата "зациклена опашка" с масив, който при нужда удвоява размера си. Имплементирайте необходимите методи за добавяне към опашката, извличане на елемента, който е наред и поглеждане на елемента, който е наред, без да го премахвате от опашката. При невалидна операция подавайте подходящо изключение.

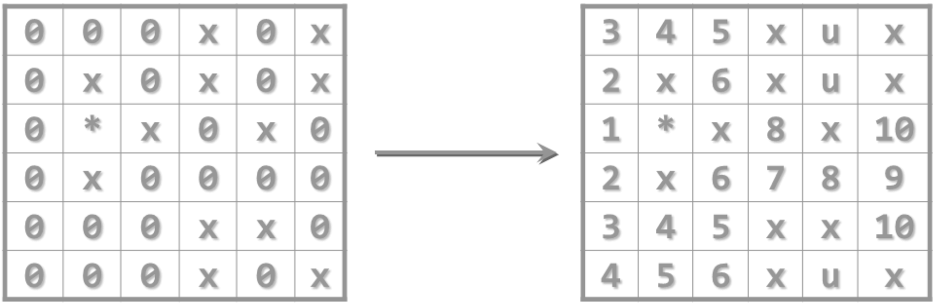
15. Реализирайте сортиране на числа в динамичен свързан списък, без да използвате допълнителен масив.

16. Използвайки опашка реализирайте пълно обхождане на всички директории на твърдия ви диск и ги отпечатвайте на конзолата. Реализирайте алгоритъма "обхождане в ширина" – Breadth-First-Search (BFS) – може да намерите стотици статии за него в Интернет.

17. Използвайки опашка реализирайте пълно обхождане на всички директории на твърдия ви диск и ги отпечатвайте на конзолата. Реализирайте алгоритъма "обхождане в дълбочина" – Depth-First-Search (DFS) – може да намерите стотици статии за него в Интернет.

18. Даден е лабиринт с размери N x N. някои от клетките на лабиринта са празни (0) а други са запълнени (x). Можем да се движим от празна клетка до друга празна клетка, ако двете имат обща стена. При дадена начална позиция (*) изчислете и попълнете лабиринта с минималната дължина от началната позиция до всяка друга. Ако някоя клетка не може да бъде достигната я попълнете с "u”.

Пример:



0	0	0	x	0	x
0	x	0	x	0	x
0	*	x	0	x	0
0	x	0	0	0	0
0	0	0	x	x	0
0	0	0	x	0	x

3	4	5	x	u	x
2	x	6	x	u	x
1	*	x	8	x	10
2	x	6	7	8	9
3	4	5	x	x	10
4	5	6	x	u	x

Решения и упътвания

1. Вижте динамичната реализация на едносвързан списък, която разгледахме в секцията "Свързан списък".
2. Използвайте **`Stack<int>`**.
3. Вижте динамичната реализация на едносвързан списък, която разгледахме в секцията "Свързан списък".
4. Използвайте **`List<int>`**. Сортирайте списъка и след това с едно обхождане намерете началния индекс и броя елементи на най-дългата подредица от равни числа. Направете нов списък и го попълнете с толкова на брой елементи.
5. Използвайте списък. Ако текущото число е положително, го добавете в списъка, ако е отрицателно, не го добавяйте.
6. Сортирайте елементите на списъка, след което ги пребройте. Вече знаем кои елементи се срещат нечетен брой пъти. Направете нов списък и с едно обхождане на списъка добавяме само елементите, който се срещат четен брой пъти.
7. Направете си масив occurrences с размер 1001. След това обхождаме списъка с числа и за всяко число number увеличаваме съответната стойност на occurrences (occurrences[number]++). Така на всеки индекс, където стойността е различна от 0 има срещане на числото и го принтираме.
8. Използвайте списък. Сортирайте списъка и така ще получите равните числа едно до друго. Обхождаме масива като броим по колко пъти се среща всяко число. Ако в даден момент едни число се среща N/2+1, то това число е мажоранта и няма нужда да проверяваме повече. Ако след позиция N/2+1 се появи ново число (до момента не е намерен мажорант и текущото число се смени), няма нужда да проверяваме повече за мажорант – дори и в случай, че списъка е запълнен до края с текущото число, то няма как да се срещне N/2+1 пъти.

9. Използвайте опашка. В началото добавете N и след това за всяко текущо число M добавете към опашката M+1, 2*M + 1 и M+2. На всяка стъпка отпечатвайте M и ако в даден момент отпечатаните числа станат 50, спрете цикъла.

10. Използвайте структурата от данни опашка. Изваждайте елементите на нива до момента, в който стигнете до M. Пазете информация за числата, чрез които сте сигнали до текущото число. Първо в опашката сложете N. За всяко извадено число, вкарвайте 3 нови (ако числото, което сте извадили е X, вкарайте X * 2, X + 2 и X + 1). Като оптимизация на решението се постарайте да избягвате повторенията на числа в опашката.

11. Имплементирайте клас `DoubleLinkedListNode`, който има полета Previous, Next и Value.

12. Използвайте едносвързан списък (подобен на списъка от предната задача, но има само поле Next, без поле Previous.

13. Използвайте два стека с общо дъно. По този начин, ако добавяме елементи отляво на дека ще влизат в левия стек, след което ще могат да бъдат премахнати отново оттам. Аналогично за десния стек.

14. Използвайте масив. Когато стигнем до последния индекс ще добавим следващия елемент в началото на масива. За точното пресмятане на индексите използвайте остатък от делене на дължината на масива. При нужда от преоразмеряване на масива можете да го направите по аналогия с реализираното преоразмеряване в секцията "Статичен списък".

15. Използвайте просто сортиране по метода на мехурчето. Започваме от най левия елемент, като проверяваме дали е по-малък от следващия. Ако не, им сменяме местата. После сравняваме със следващия и т.н. докато достигнем до такъв, който е по-голям от него или не стигнем края на масива. Връщаме се в началото и взимаме пак първия като повтаряме процедурата. Ако първият е по-малък, взимаме следващия и започваме да сравняваме. Повтаряме тези операции докато не стигнем до момент, в който сме взели последователно всички елементи и на нито един не се наложило да бъде преместен.

16. Алгоритъмът е много лесен: започваме от празна опашка, в която слагаме коренната директория (от която стартира обхождането). След това докато опашката не остане празна, изваждаме от нея поредната директория, отпечатваме я и прибавяме към опашката всички нейни поддиректории. По този начин ще обходим файловата система в ширина. Ако в нея няма цикли (както е под Windows), процесът ще е краен.

17. Ако в решението на предната задача заместим опашката със стек, ще получим обхождане в дълбочина.

18. Използвайте обхождане в ширина (Breath-First Search), като започваме обхождането от позицията маркирана с ‘*’. В всяка непосетена съседна клетка на текущата клетка, записваме текущото число + 1, като приемаме, че стойността на ‘*’ е 0. След като опашката се изпразни, обхождаме цялата матрица и ако на някоя клетка имаме 0, записваме стойност ‘u’.

Глава 17. Дървета и графи

В тази тема...

В настоящата тема ще разгледаме т. нар. дървовидни структури от данни, каквито са дърветата и графите. Познаването на свойствата на тези структури е важно за съвременното програмиране. Всяка от тях се използва за моделирането на проблеми от реалността, които се решават ефективно с тяхна помощ. Ще разгледаме в детайли какво представляват дървовидните структури от данни и ще покажем техните основни предимства и недостатъци. Ще дадем примерни реализации и задачи, демонстриращи реалната им употреба. Ще се спрем по-подробно на двоичните дървета, наредените двоични дървета за претърсване и балансираните дървета. Ще разгледаме структурата от данни "граф", видовете графи и тяхната употреба. Ще покажем и къде във .NET Framework се използват имплементации на балансирани дървета за търсене.

Дървовидни структури

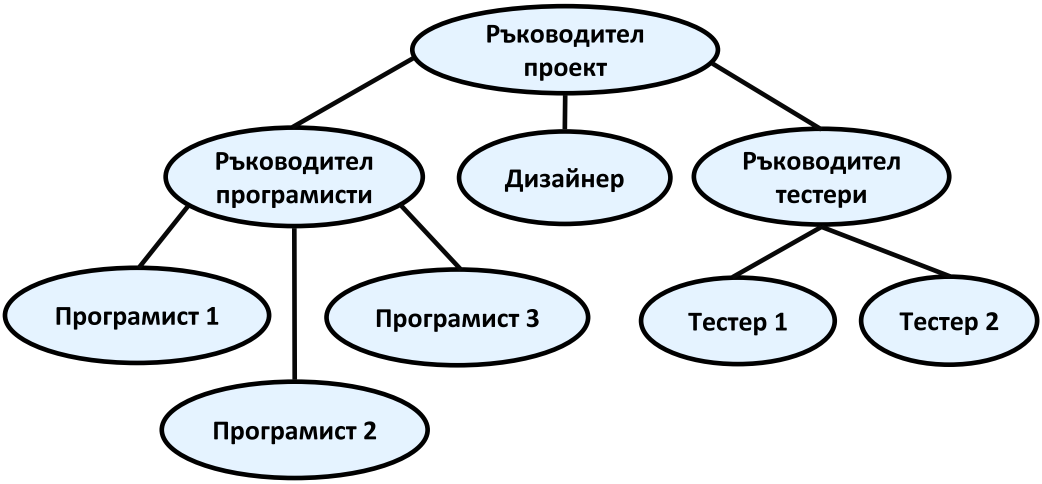
В много ситуации в ежедневието се налага да опишем (моделираме) съвкупност от обекти, които са взаимно свързани по някакъв начин и то така, че не могат да бъдат описани чрез досега изложените линейни структури от данни. В следващите няколко точки от тази тема ще дадем примери за такива структури, ще покажем техните свойства и съответно практическите задачи, които са довели до тяхното възникване.

Дървета

В програмирането дърветата са изключително често използвана структура от данни, защото те моделират по естествен начин всякакви йерархии от обекти, които постоянно ни заобикалят в реалния свят. Нека дадем един пример, преди да изложим терминологията, свързана с дърветата.

Пример – йерархия на участниците в един софтуерен проект

Да вземем за пример един екип, отговорен за изработването на даден софтуерен проект. Участниците в него са взаимно свързани с връзката ръководител-подчинен. Ще разгледаме една конкретна ситуация, в която имаме екип от 9 души:

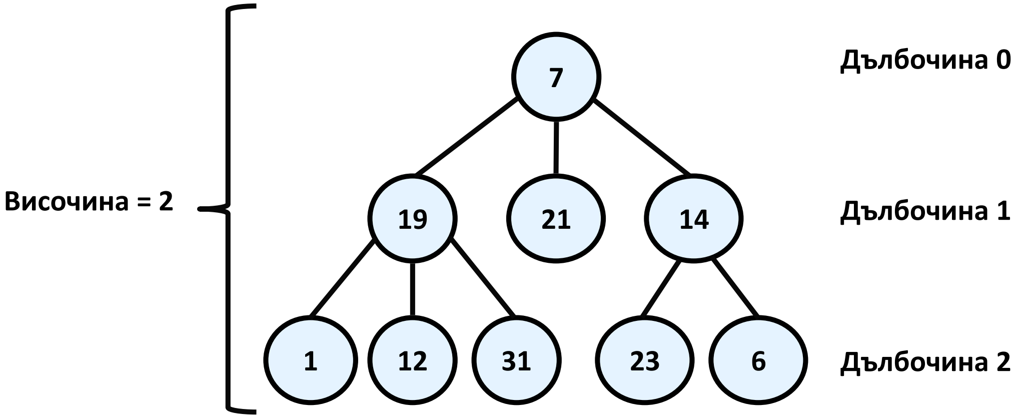


Каква информация можем да извлечем от така изобразената йерархия? Прекият шеф на програмистите е съответно "Ръководител програмисти". "Ръководител проект" е също е техен началник, но непряк, т.е. те отново са му подчинени. "Ръководител програмисти" е подчинен само на "Ръководител проект". От друга страна, ако погледнем "Програмист 1", той няма нито един подчинен. "Ръководител проект" стои най-високо в йерархията и няма шеф.

По аналогичен начин можем да опишем и ситуацията с останалите участници в проекта. Виждаме как една на пръв поглед малка фигура ни носи много информация.

Терминология свързана с дърветата

За по-доброто разбиране на тази точка силно препоръчваме на читателя да се опита на всяка стъпка да прави аналогия между тяхното абстрактно значение и това, което използваме в ежедневието.



Нека да опростим начина, по който изобразихме нашата йерархия. Можем да приемем, че тя се състои от точки, свързани с отсечки. За удобство, точките ще номерираме с произволни числа, така че после лесно да можем да говорим за някоя конкретна.

Всяка една точка, ще наричаме **връх**, а всяка една отсечка – **ребро**. Върховете "19", "21" и "14" стоят под върха "7" и са директно свързани с него. Тях ще наричаме **преки наследници (деца)** на "7", а "7" – техен **родител (баща)**. Аналогично "1", "12" и "31" са деца на "19" и "19" е техен родител. Съвсем естествено ще казваме, че "21" е **брат** на "19", тъй като са деца на "7" (обратното също е вярно – "19" е брат на "21"). От гледна точка на "1", "12", "31", "23" и "6", "7" е предшестващ ги в йерархията (в случая е родител на техните родители). Затова "7" ще наречем техен **непряк предшественик (дядо, прародител)**, а тях – негови **непреки наследници**.

Корен е върхът, който няма предшественици. В нашия случай той е "7".

Листа са всички върхове, които нямат наследници. В примера – "1", "12", "31", "21", "23" и "6" са листа.

Вътрешни върхове са всички върхове, различни от корена и листата (т.е. всички върхове, които имат както родител, така и поне един наследник). Такива са "19" и "14".

Път ще наричаме последователност от свързани чрез ребра върхове, в която няма повтарящи се върхове. Например последователността "1", "19", "7" и "21" е път. "1", "19" и "23" не е път, защото "19" и "23" не са свързани помежду си с ребро.

Дължина на път е броят на ребрата, свързващи последователността от върхове в пътя. Практически този брой е равен на броят на върховете в пътя минус единица. Дължината на примера ни за път ("1", "19", "7" и "21") е три.

Дълбочина на връх ще наричаме дължината на пътя от корена до дадения връх. На примера ни "7" като корен е с дълбочина нула, "19" е с дълбочина едно, а "23" – с дълбочина две.

И така, ето и дефиницията за това какво е дърво:

Дърво (tree) –рекурсивна структура от данни, която се състои от върхове, които са свързани помежду си с ребра. За дърветата са в сила твърденията:

- Всеки връх може да има 0 или повече преки наследници (деца).
- Всеки връх има най-много един баща. Съществува точно един специален връх, който няма предшественици – коренът (ако дървото не е празно).
- Всички върхове са достижими от корена, т.е съществува път от корена до всички тях.

Можем да дефинираме дърво и по по-прост начин: всеки единичен връх наричаме дърво и той може да има нула или повече наследници, които също са дървета.

Височина на дърво е максималната от дълбочините на всички върхове. В горния пример височината е 2.

Степен на връх ще наричаме броят на преките наследници (деца) на дадения връх. Степента на "19" и "7" е три, докато тази на "14" е две. Листата са от нулева степен.

Разклоненост на дърво се нарича максималната от степените на всички върхове в дървото. В нашият пример степента на върховете е най-много 3, следователно разклонеността на дървото ни е 3.

Реализация на дърво – пример

Нека сега разгледаме как можем да представяме дърветата като структури от данни в програмирането. Ще реализираме дърво, което съдържа числа във върховете си и всеки връх може да има 0 или повече наследници, които също са дървета (следвайки рекурсивната дефиниция). Всеки връх от дървото е рекурсивно-дефиниран чрез себе си. Един връх от дървото (**`TreeNode<T>`**) съдържа в себе си списък от наследници, които също са върхове от дървото (**`TreeNode<T>`**). Нека разгледаме сорс кода:

```
using System;
using System.Collections.Generic;

/// <summary>
/// Represents a tree node
/// </summary>
/// <typeparam name="T">the type of the values in
/// nodes</typeparam>
public class TreeNode<T>
{
    // Contains the value of the node
    private T value;

    // Shows whether the current node has parent
    private bool hasParent;

    // Contains the children of the node
    private List<TreeNode<T>> children;

    /// <summary>
    /// Constructs a tree node
    /// </summary>
    /// <param name="value">the value of the node</param>
    public TreeNode(T value)
    {
        if (value == null)
        {
            throw new ArgumentNullException(
                "Cannot insert null value!");
        }
        this.value = value;
        this.children = new List<TreeNode<T>>();
    }

    /// <summary>
    /// The value of the node
    /// </summary>
    public T Value
    {
        get
        {
            return this.value;
        }
        set
        {
            this.value = value;
        }
    }

    /// <summary>
    /// The number of node's children
    /// </summary>
    public int ChildrenCount
    {
        get
        {
            return this.children.Count;
        }
    }

    /// <summary>
    /// Adds child to the node
    /// </summary>
    /// <param name="child">the child to be added</param>
    public void AddChild(TreeNode<T> child)
    {
        if (child == null)
        {
            throw new ArgumentNullException(
                "Cannot insert null value!");
        }

        if (child.hasParent)
        {
            throw new ArgumentException(
                "The node already has a parent!");
        }

        child.hasParent = true;
        this.children.Add(child);
    }

    /// <summary>
    /// Gets the child of the node at given index
    /// </summary>
    /// <param name="index">the index of the desired child</param>
    /// <returns>the child on the given position</returns>
    public TreeNode<T> GetChild(int index)
    {
        return this.children[index];
    }
}

/// <summary>
/// Represents a tree data structure
/// </summary>
/// <typeparam name="T">the type of the values in the
/// tree</typeparam>
public class Tree<T>
{
    // The root of the tree
```

```
private TreeNode<T> root;

/// <summary>
/// Constructs the tree
/// </summary>
/// <param name="value">the value of the node</param>
public Tree(T value)
{
   if (value == null)
   {
      throw new ArgumentNullException(
         "Cannot insert null value!");
   }

   this.root = new TreeNode<T>(value);
}

/// <summary>
/// Constructs the tree
/// </summary>
/// <param name="value">the value of the root node</param>
/// <param name="children">the children of the root
/// node</param>
public Tree(T value, params Tree<T>[] children)
   : this(value)
{
   foreach (Tree<T> child in children)
   {
      this.root.AddChild(child.root);
   }
}

/// <summary>
/// The root node or null if the tree is empty
/// </summary>
public TreeNode<T> Root
{
   get
   {
      return this.root;
   }
}

/// <summary>
/// Traverses and prints tree in Depth
/// First Search (DFS) manner
/// </summary>
/// <param name="root">the root of the tree to be
/// traversed</param>
/// <param name="spaces">the spaces used for
```

```
   /// representation of the parent-child relation</param>
   private void PrintDFS(TreeNode<T> root, string spaces)
   {
      if (this.root == null)
      {
         return;
      }

      Console.WriteLine(spaces + root.Value);

      TreeNode<T> child = null;
      for (int i = 0; i < root.ChildrenCount; i++)
      {
         child = root.GetChild(i);
         PrintDFS(child, spaces + "   ");
      }
   }

   /// <summary>
   /// Traverses and prints the tree in
   /// Depth First Search (DFS) manner
   /// </summary>
   public void PrintDFS()
   {
      this.PrintDFS(this.root, string.Empty);
   }
}

/// <summary>
/// Shows a sample usage of the Tree<T> class
/// </summary>
public static class TreeExample
{
   static void Main()
   {
      // Create the tree from the sample
      Tree<int> tree =
         new Tree<int>(7,
            new Tree<int>(19,
               new Tree<int>(1),
               new Tree<int>(12),
               new Tree<int>(31)),
            new Tree<int>(21),
            new Tree<int>(14,
               new Tree<int>(23),
               new Tree<int>(6))
         );

      // Traverse and print the tree using Depth-First-Search
      tree.PrintDFS();
```

```
        // Console output:
        // 7
        //    19
        //        1
        //        12
        //        31
        //    21
        //    14
        //        23
        //        6
    }
}
```

Как работи нашата имплементация на дърво?

Нека кажем няколко думи за предложения код. В примера имаме клас **`Tree<T>`**, който е имплементация на самото дърво. Дефиниран е и клас – **`TreeNode<T>`**, който представлява един връх от дървото.

Функционалността, свързана с връх, като например създаване на връх, добавяне на наследник на връх, взимане на броя на наследниците и т.н. се реализират на ниво **`TreeNode<T>`**.

Останалата функционалност (например обхождане на дървото) се реализира на ниво **`Tree<T>`**. Така функционалността става логически разделена между двата класа, което прави имплементацията по гъвкава.

Причината да разделим на два класа имплементацията е, че някои операции се отнасят за конкретен връх (например добавяне на наследник), докато други се отнасят за цялото дърво (например търсене на връх по неговата стойност). При такова разделяне дървото е клас, който знае кой му е коренът, а всеки връх знае наследниците си. При такава имплементация е възможно да имаме и празно дърво (при **`root=null`**).

Ето и някои подробности от реализацията на **`TreeNode<T>`**. Всеки един връх (node) на дървото представлява съвкупност от частно поле **`value`**, което съдържа стойността му, и списък от наследници **`children`**. Списъкът на наследниците е от елементи на същия тип. Така всеки връх съдържа списък от референции към неговите преки наследници. Предоставени са също и публични свойства за достъп до стойността на върха. Операциите, които могат да се извършват от външен за класа код върху децата, са:

- **`AddChild(TreeNode<T> child)`** – добавя нов наследник.
- **`TreeNode<T> GetChild(int index)`** – връща наследник по зададен индекс.
- **`ChildrenCount`** – връща броя на наследници на даден връх.

За да спазим изискването всеки връх в дървото да има точно един родител, сме дефинирали частното поле **`hasParent`**, което показва дали даденият връх има родител. Тази информация се използва вътрешно в нашия клас и ни трябва в метода `AddChild(Tree<T> child)`. В него правим проверка дали кандидат детето няма вече родител. Ако има, се хвърля изключение, показващ, че това е недопустимо.

В класа `Tree<T>` сме предоставили едно единствено **get** свойство - `TreeNode<T> Root`, което връща корена на дървото.

Рекурсивно обхождане на дърво в дълбочина

В класа `Tree<T>` е реализиран и методът `TraverseDFS()`, който извиква частния метод `PrintDFS(TreeNode<T> root, string spaces)`, който обхожда дървото в дълбочина и извежда на стандартния изход елементите му, така че нагледно да се изобрази дървовидната структура чрез отместване надясно (с добавяне на интервали).

Алгоритъмът за **обхождане в дълбочина** (**Depth-First-Search** или **DFS**) започва от даден връх и се стреми да се спусне колкото се може по-надолу в дървовидната йерархия. Когато стигне до връх, от който няма продължение се връща назад към предходния връх. Алгоритъма можем да опишем схематично по следния начин:

1. Обхождаме текущия връх.
2. Последователно обхождаме рекурсивно всяко едно от поддърветата на текущия връх (обръщаме се рекурсивно към същия метод последователно за всеки един от неговите преки наследници).

Създаване на дърво

За да създаваме по-лесно дървета сме дефинирали специален конструктор, който приема стойност на връх и списък от поддървета за този връх. Така позволяваме подаването на произволен брой аргументи от тип `Tree<T>` (поддървета). При създаването на дървото за нашия пример използваме точно този конструктор и той ни позволява да онагледим структурата му.

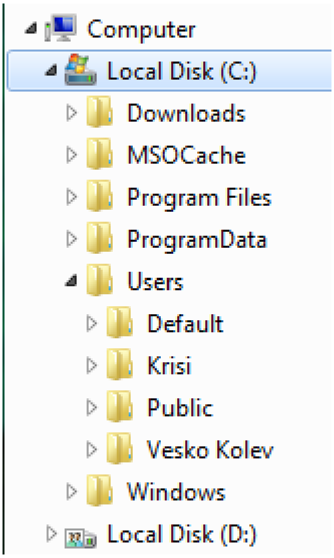
Обхождане на директориите по твърдия диск

Нека сега разгледаме още един пример за дърво – файловата система. Замисляли ли сте се, че директориите върху твърдия ви диск образуват йерархична структура, която е дърво? Можете да се сетите и за много други реални примери, при които се използват дървета.

Нека разгледаме по-подробно файловата система в Windows. Както знаем от нашия всекидневен опит, ние създаваме папки върху твърдия диск, които могат да съдържат от своя страна подпапки или файлове. Подпапките отново може да съдържат подпапки и т. н. до някакво разумно ограничение (максимална дълбочина).

Дървото на директориите на файловата система е достъпно чрез стандартни функции от класа `System.IO.DirectoryInfo`. То не съществува като структура от данни в явен вид, но съществува начин да извличаме за всяка директория файловете и директориите в нея и следователно можем да го обходим чрез стандартен алгоритъм за обхождане на дървета.

Ето как изглежда типичното дърво на директориите в Windows:


Computer
Local Disk (C:)
Downloads
MSOCache
Program Files
ProgramData
Users
Default
Krisi
Public
Vesko Kolev
Windows
Local Disk (D:)

Рекурсивно обхождане на директориите в дълбочина

Следващият пример показва как да обходим рекурсивно (в дълбочина, по алгоритъма Depth-First-Search) дървовидната структура на дадена папка и да изведем на стандартния изход нейното съдържание:

DirectoryTraverserDFS.cs

```
using System;
using System.IO;

/// <summary>
/// Sample class, which traverses recursively given directory
/// based on the Depth-First-Search (DFS) algorithm
/// </summary>
public static class DirectoryTraverserDFS
{
   /// <summary>
   /// Traverses and prints given directory recursively
   /// </summary>
   /// <param name="dir">the directory to be traversed</param>
   /// <param name="spaces">the spaces used for representation
   /// of the parent-child relation</param>
   private static void TraverseDir(DirectoryInfo dir,
      string spaces)
   {
    // Visit the current directory
    Console.WriteLine(spaces + dir.FullName);

    DirectoryInfo[] children = dir.GetDirectories();

    // For each child go and visit its subtree
    foreach (DirectoryInfo child in children)
    {
      TraverseDir(child, spaces + "  ");
    }
  }

  /// <summary>
  /// Traverses and prints given directory recursively
  /// </summary>
  /// <param name="directoryPath">the path to the directory
  /// which should be traversed</param>
  public static void TraverseDir(string directoryPath)
  {
    TraverseDir(new DirectoryInfo(directoryPath),
      string.Empty);
  }

  public static void Main()
  {
    TraverseDir("C:\\");
  }
}
```

Както се вижда от примера, рекурсивното обхождане на съдържанието на директория по нищо не се различава от обхождането на нашето дърво.

Ето как изглежда резултатът от обхождането (със съкращения):

```
C:\
  C:\Config.Msi
  C:\Documents and Settings
    C:\Documents and Settings\Administrator
      C:\Documents and Settings\Administrator\.ARIS70
      C:\Documents and Settings\Administrator\.jindent
      C:\Documents and Settings\Administrator\.nbi
        C:\Documents and Settings\Administrator\.nbi\downloads
        C:\Documents and Settings\Administrator\.nbi\log
        C:\Documents and Settings\Administrator\.nbi\cache
        C:\Documents and Settings\Administrator\.nbi\tmp
        C:\Documents and Settings\Administrator\.nbi\wd
      C:\Documents and Settings\Administrator\.netbeans
        C:\Documents and Settings\Administrator\.netbeans\6.0
...
```

Обхождане на директориите в ширина

Нека сега разгледаме още един начин да обхождаме дървета. **Обхождането в ширина** (**Breath-First-Search** или **BFS**) е алгоритъм за обхождане на дървовидни структури от данни, при който първо се посещава началния връх, след това неговите преки деца, след тях преките деца на децата и т.н. Този процес се нарича **метод на вълната**, защото прилича на вълните, образувани от камък, хвърлен в езеро.

Алгоритъмът за обхождане на дърво в ширина по метода на вълната можем да опишем схематично по следния начин:

1. Записваме в опашката `Q` началния връх.
2. Докато `Q` не е празна повтаряме следните две стъпки:
   - Изваждаме от `Q` поредния връх **`v`** и го отпечатваме.
   - Добавяме всички наследници на **`v`** в опашката.

Алгоритъмът BFS е изключително прост и има свойството да обхожда първо най-близките до началния връх върхове, след тях по-далечните и т.н. и най-накрая – най-далечните върхове. С времето ще се убедите, че BFS алгоритъмът има широко приложение при решаването на много задачи, като например при търсене на най-кратък път в лабиринт.

Нека сега приложим BFS алгоритъма за отпечатване на всички директории от файловата система:

DirectoryTraverserBFS.cs

```
using System;
using System.Collections.Generic;
using System.IO;

/// <summary>
/// Sample class, which traverses given directory
/// based on the Breath-First-Search (BFS) algorithm
/// </summary>
public static class DirectoryTraverserBFS
{
   /// <summary>
   /// Traverses and prints given directory with BFS
   /// </summary>
   /// <param name="directoryPath">the path to the directory
   /// which should be traversed</param>
   public static void TraverseDir(string directoryPath)
   {
      Queue<DirectoryInfo> visitedDirsQueue =
         new Queue<DirectoryInfo>();
      visitedDirsQueue.Enqueue(new DirectoryInfo(directoryPath));
      while (visitedDirsQueue.Count > 0)
      {
```

```
      DirectoryInfo currentDir = visitedDirsQueue.Dequeue();
      Console.WriteLine(currentDir.FullName);

      DirectoryInfo[] children = currentDir.GetDirectories();
      foreach (DirectoryInfo child in children)
      {
        visitedDirsQueue.Enqueue(child);
      }
    }
  }

  public static void Main()
  {
    TraverseDir("C:\\ ");
  }
}
```

Ако стартираме програмата, ще се убедим, че обхождането в ширина първо открива най-близките директории до корена (дълбочина 1), след тях всички директории на дълбочина 2, след това директориите на дълбочина 3 и т.н. Ето примерен изход от програмата:

```
C:\
C:\Config.Msi
C:\Documents and Settings
C:\Inetpub
C:\Program Files
C:\RECYCLER
C:\System Volume Information
C:\WINDOWS
C:\wmpub
C:\Documents and Settings\Administrator
C:\Documents and Settings\All Users
C:\Documents and Settings\Default User
...
```

Двоични дървета

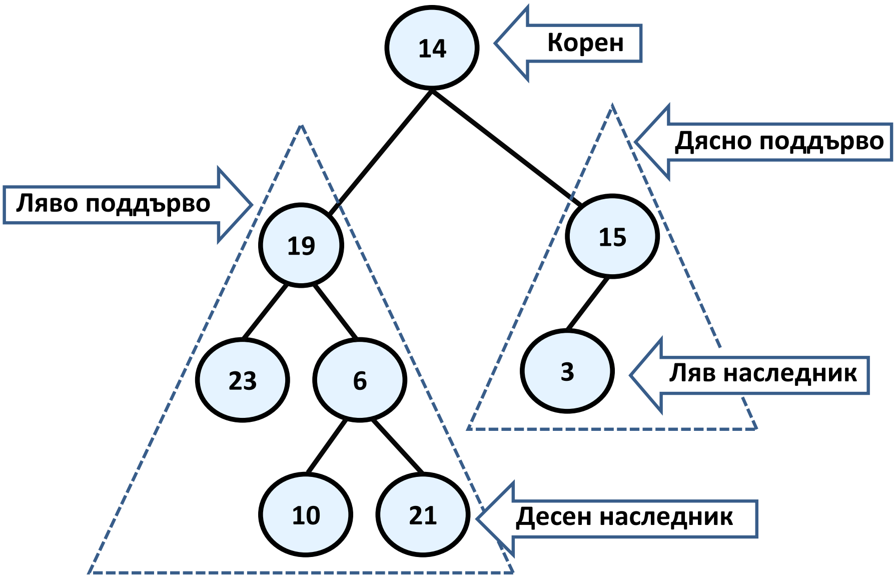
В предишната точка от темата разгледахме обобщената структура дърво. Сега ще преминем към един неин полезен частен случай, който се оказва изключително важен за практиката – **двоично дърво**. Важно е да отбележим, че термините, които дефинирахме до момента, важат с пълна сила и при този вид дърво. Въпреки това, по-долу ще дадем и някои допълнителни, специфични за дадената структура определения.

Двоично дърво (binary tree) – дърво, в което всеки връх е от степен не надвишаваща две т.е. дърво с разклоненост две. Тъй като преките наследници (деца) на всеки връх са най-много два, то е прието да се въвежда наредба между тях, като единият се нарича **ляв наследник**, а другият – **десен наследник**. Те, от своя страна, са корени съответно на **лявото поддърво** и на **дясното поддърво** на техния родител.

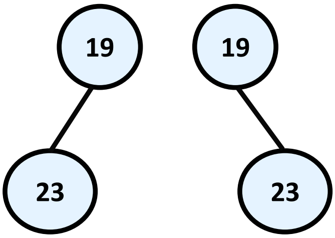
Двоично дърво – пример

Ето и едно примерно двоично дърво, което ще използваме за изложението по-нататък. В този пример отново въвеждаме номерация на върховете, която е абсолютно произволна и която ще използваме, за да може по-лесно да говорим за всеки връх.

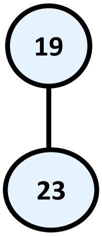
На примера са изобразени съответно корена на дървото "14", пример за ляво поддърво (с корен "19") и дясно поддърво (с корен "15"), както и ляв и десен наследник – съответно "3" и "21".



Следва да отбележим обаче, че двоичните дървета имат едно много сериозно различие в дефиницията си, за разлика от тази на обикновеното дърво – наредеността на наследниците на всеки връх. Следващият пример ясно показва това различие:



На схемата са изобразени две абсолютно различни **двоични дървета** – в единия случай коренът е "19" и има **ляв наследник** "23", а в другия имаме двоично дърво с корен отново "19", но с "23" за **десен наследник**. Ако разгледаме обаче двете структури като **обикновени дървета**, те ще са абсолютно еднакви и неразличими. Затова такова **дърво** бихме изобразили по следния начин:



Запомнете! Въпреки, че разглеждаме двоичните дървета като подмножество на структурата дърво, трябва да се отбележи, че условието за нареденост на наследниците ги прави до голяма степен различни като структури.

Обхождане на двоично дърво

Обхождането на дърво по принцип е една класическа и често срещана задача. В случая на двоичните дървета има няколко основни начина за обхождане:

- **ЛКД (Ляво-Корен-Дясно/Inorder)** – обхождането става като първо се обходи лявото поддърво, след това корена и накрая дясното поддърво. В нашият пример последователността, която се получава при обхождането е: "23", "19", "10", "6", "21", "14", "3", "15".
- **КЛД (Корен-Ляво-Дясно/Preorder)** – в този случай първо се обхожда корена на дървото, после лявото поддърво и накрая дясното. Ето и как изглежда резултатът от този вид обхождане: "14", "19", "23", "6", "10", "21", "15", "3".
- **ЛДК (Ляво-Дясно-Корен/Postorder)** – тук по аналогичен на горните два примера начин, обхождаме първо лявото поддърво, после дясното и накрая коренът. Резултатът след обхождането е "23", "10", "21", "6", "19", "3", "15", "14".

Обхождане на двоично дърво с рекурсия – пример

В следващия пример ще покажем примерна реализация на двоично дърво, което ще обходим по схемата ЛКД:

```
using System;
using System.Collections.Generic;

/// <summary>
/// Represents a binary tree node
/// </summary>
/// <typeparam name="T">the type of the values in
nodes</typeparam>
```

```
public class BinaryTreeNode<T>
{
   // Contains the value of the node
   private T value;

   // Shows whether the current node has parent
   private bool hasParent;

   // Contains the left child of the node
   private BinaryTreeNode<T> leftChild;

   // Contains the right child of the node
   private BinaryTreeNode<T> rightChild;

   /// <summary>
   /// Constructs a binary tree node
   /// </summary>
   /// <param name="value">the value of the node</param>
   /// <param name="leftChild">the left child of the node</param>
   /// <param name="rightChild">the right child of the
   /// node</param>
   public BinaryTreeNode(T value,
        BinaryTreeNode<T> leftChild,
        BinaryTreeNode<T> rightChild)
   {
      if (value == null)
      {
         throw new ArgumentNullException(
            "Cannot insert null value!");
      }

      this.value = value;
      this.LeftChild = leftChild;
      this.RightChild = rightChild;
   }

   /// <summary>
   /// Constructs a binary tree node with no children
   /// </summary>
   /// <param name="value">the value of the node</param>
   public BinaryTreeNode(T value) : this(value, null, null)
   {
   }

   /// <summary>
   /// The value of the node
   /// </summary>
   public T Value
   {
      get
```

```
        {
            return this.value;
        }
        set
        {
            this.value = value;
        }
    }

    /// <summary>
    /// The left child of the node
    /// </summary>
    public BinaryTreeNode<T> LeftChild
    {
        get
        {
            return this.leftChild;
        }
        set
        {
            if (value == null)
            {
                return;
            }

            if (value.hasParent)
            {
                throw new ArgumentException(
                    "The node already has a parent!");
            }

            value.hasParent = true;
            this.leftChild = value;
        }
    }

    /// <summary>
    /// The right child of the node
    /// </summary>
    public BinaryTreeNode<T> RightChild
    {
        get
        {
            return this.rightChild;
        }
        set
        {
            if (value == null)
            {
                return;
```

```
         }

         if (value.hasParent)
         {
            throw new ArgumentException(
               "The node already has a parent!");
         }

         value.hasParent = true;
         this.rightChild = value;
      }
   }
}

/// <summary>
/// Represents a binary tree structure
/// </summary>
/// <typeparam name="T">the type of the values in the
/// tree</typeparam>
public class BinaryTree<T>
{
   // The root of the tree
   private BinaryTreeNode<T> root;

   /// <summary>
   /// Constructs the tree
   /// </summary>
   /// <param name="value">the value of the root node</param>
   /// <param name="leftChild">the left child of the root
node</param>
   /// <param name="rightChild">the right child of the
   /// root node</param>
   public BinaryTree(T value, BinaryTree<T> leftChild,
         BinaryTree<T> rightChild)
   {
      if (value == null)
      {
         throw new ArgumentNullException(
            "Cannot insert null value!");
      }

      BinaryTreeNode<T> leftChildNode =
         leftChild != null ? leftChild.root : null;

      BinaryTreeNode<T> rightChildNode =
         rightChild != null ? rightChild.root : null;

      this.root = new BinaryTreeNode<T>(
         value, leftChildNode, rightChildNode);
   }
```

```
/// <summary>
/// Constructs the tree
/// </summary>
/// <param name="value">the value of the root node</param>
public BinaryTree(T value)
   : this(value, null, null)
{
}

/// <summary>
/// The root of the tree.
/// </summary>
public BinaryTreeNode<T> Root
{
   get
   {
      return this.root;
   }
}

/// <summary>
/// Traverses binary tree in pre-order manner
/// </summary>
/// <param name="root">the binary tree to be traversed</param>
private void PrintInorder(BinaryTreeNode<T> root)
{
   if (root == null)
   {
      return;
   }

   // 1. Visit the left child
   PrintInorder(root.LeftChild);

   // 2. Visit the root of this subtree
   Console.Write(root.Value + " ");

   // 3. Visit the right child
   PrintInorder(root.RightChild);
}

/// <summary>
/// Traverses and prints the binary
/// tree in pre-order manner
/// </summary>
public void PrintInorder()
{
   PrintInorder(this.root);
   Console.WriteLine();
   }
}

/// <summary>
/// Shows how the BinaryTree class can be used
/// </summary>
public class BinaryTreeExample
{
   public static void Main()
   {
      // Create the binary tree from the sample
      BinaryTree<int> binaryTree =
         new BinaryTree<int>(14,
               new BinaryTree<int>(19,
                     new BinaryTree<int>(23),
                     new BinaryTree<int>(6,
                           new BinaryTree<int>(10),
                           new BinaryTree<int>(21))),
               new BinaryTree<int>(15,
                     new BinaryTree<int>(3),
                     null));

      // Traverse and print the tree in in-order manner
      binaryTree.PrintInorder();

      // Console output:
      // 23 19 10 6 21 14 3 15
   }
}
```

Как работи примерът?

Тази примерна имплементация на двоично дърво не се различава съществено от реализацията, която показахме в случая на обикновено дърво. Отново имаме отделни класове за представяне на двоично дърво и на връх в такова – **BinaryTree<T>** и **BinaryTreeNode<T>**. В класа **BinaryTreeNode<T>** имаме частни полета **value** и **hasParent**. Както и преди, първото съдържа стойността на върха, а второто показва дали върха има родител. При добавяне на ляв или десен наследник (ляво/дясно дете) на даден връх, се прави проверка дали имат вече родител и ако имат, се хвърля изключение, аналогично на реализацията ни на дърво.

За разлика от реализацията на обикновеното дърво, сега вместо списък на децата, всеки връх съдържа по едно частно поле за ляв и десен наследник. За всеки от тях сме дефинирали публични свойства, за да могат да се достъпват от външен за класа код.

В **BinaryTree<T>** е реализирано едно единствено **get** свойство, което връща корена на дървото. Методът **PrintInorder()** извиква вътрешно метода `PrintInorder(BinaryTreeNode<T> root)`. Вторият метод, от своя страна, обхожда подаденото му дърво по схемата ляво-корен-дясно (ЛКД). Това става по следния тристъпков алгоритъм:

1. Рекурсивно извикване на метода за обхождане за лявото поддърво на дадения връх.
2. Обхождане на самия връх.
3. Рекурсивно извикване на метода за обхождане на дясното поддърво.

Силно препоръчваме на читателя да се опита (като едно добро упражнение) да модифицира предложения алгоритъм и код самостоятелно, така че да реализира другите два основни типа обхождане.

Наредени двоични дървета за претърсване

До момента видяхме как можем да построим обикновено дърво и двоично дърво. Тези структури сами по себе си са доста обобщени и трудно, в такъв суров вид, могат да ни свършат някаква по-сериозна работа. На практика в информатиката се прилагат някои техни разновидности, в които са дефинирани съвкупност от строги правила (алгоритми) за различни операции с тях и с техните елементи. Всяка една от тези разновидности носи със себе си специфични свойства, които са полезни в различни ситуации.

Като примери за такива полезни свойства могат да се дадат бързо търсене на елемент по зададена стойност (червено-черно дърво); нареденост (сортираност) на елементите в дървото; възможност да се организира голямо количество информация на някакъв файлов носител, така че търсенето на елемент в него да става бързо с възможно най-малко стъпки (B-дърво), както и много други.

В тази секция ще разгледаме един по-специфичен клас двоични дървета – **наредените**. Те използват едно често срещано при двоичните дървета свойство на върховете, а именно съществуването на **уникален идентификационен ключ** във всеки един от тях. Този ключ не се среща никъде другаде в рамките на даденото дърво. Друго основно свойство на тези ключове е, че са сравними. Наредените двоични дървета позволяват бързо (в общия случай с приблизително log(n) на брой стъпки) търсене, добавяне и изтриване на елемент, тъй като поддържат елементите си индиректно в сортиран вид.

Сравнимост между обекти

Преди да продължим, ще въведем следната дефиниция, от която ще имаме нужда в по-нататъшното изложение.

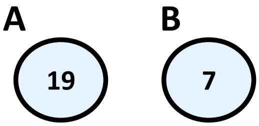
Сравнимост – два обекта A и B наричаме сравними, ако е изпълнена **точно една** от следните три зависимости между тях:

- "A е по-малко от B"
- "A е по-голямо от B"

- "A е равно на B"

Аналогично два ключа A и B ще наричаме сравними, ако е изпълнена точно една от следните три възможности: A < B, A > B или A = B.

Върховете на едно дърво могат да съдържат най-различни полета. В понататъшното разсъждение ние ще се интересуваме само от техните уникални ключове, които ще искаме да са сравними. Да покажем един пример. Нека са дадени два конкретни върха A и B:



В примера ключът на A и B са съответно целите числа 19 и 7. Както знаем от математиката, целите числа (за разлика от комплексните например) са **сравними**, което според гореизложените разсъждения ни дава правото да ги използваме като ключове. Затова за върховете A и B можем да кажем, че "A е по-голямо от B" тъй като "19 е по-голямо от 7".

⚠	**Забележете! Този път числата изобразени във върховете са техни уникални идентификационни ключове, а не както досега произволни числа.**

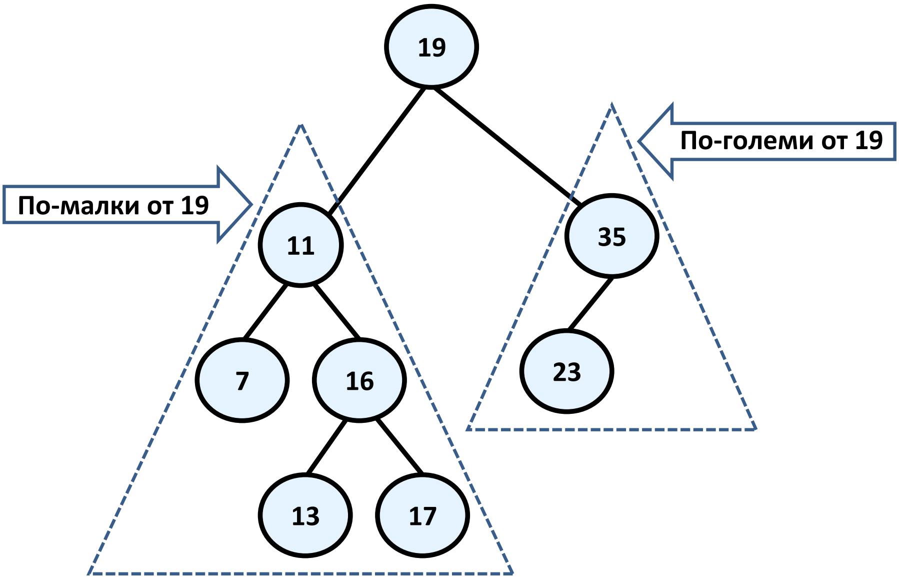
Стигаме и до дефиницията за **наредено двоично дърво за търсене:**

Наредено двоично дърво (дърво за търсене, binary search tree) е двоично дърво, в което всеки връх има уникален ключ, всеки два от ключовете са сравними и което е организирано така, че за всеки връх да е изпълнено:

- Всички ключове в **лявото** му поддърво са **по-малки** от неговия ключ.
- Всички ключове в **дясното** му поддърво са **по-големи** от неговия ключ.

Свойства на наредените двоични дървета за претърсване

На фигурата е изобразен пример за наредено двоично дърво за претърсване. Ще използваме този пример, за да дадем някои важни свойства на наредеността на двоично дърво:



По дефиниция имаме, че лявото поддърво на всеки един от върховете се състои само от елементи, които са по-малки от него, докато в дясното поддърво има само по-големи елементи. Това означава, че ако искаме да намерим даден елемент тръгвайки от корена, то или сме го намерили или трябва да го търсим съответно в лявото или дясното му поддърво, с което ще спестим излишни сравнения. Например, ако търсим в нашето дърво 23, то няма смисъл да го търсим в лявото поддърво на 19, защото 23 със сигурност не е там (23 е по-голямо от 19 следователно евентуално е в дясното поддърво). Това ни спестява 5 излишни сравнения с всеки един от елементите от лявото поддърво, които, ако използваме свързан списък, например, ще трябва да извършим.

От наредеността на елементите следва, че **най-малкият** елемент в дървото е най-левият наследник на корена, ако има такъв, или самият корен, ако той няма ляв наследник. По абсолютно същия начин **най-големият** елемент в дървото е най-десният наследник на корена, а ако няма такъв – самият корен. В нашия пример това са минималният елемент 7 и максималният – 35. Полезно и директно следващо свойство от това е, че всеки един елемент от лявото поддърво на даден връх е по-малък от всеки друг, който е в дясното поддърво на същия връх.

Наредени двоични дървета за търсене – пример

Следващият пример показва реализация на двоично дърво за търсене. Целта ни ще бъде да предложим методи за добавяне, търсене и изтриване на елемент в дървото. За всяка една от тези операции ще дадем подробно обяснение как точно се извършва.

Наредени двоични дървета: реализация на върховете

Както и преди, сега ще дефинираме вътрешен клас, който да опише структурата на един връх. По този начин ясно ще разграничим и капсулираме структурата на един връх като същност, която дървото ни ще съдържа в себе си. Този отделен клас сме дефинирали като частен и е видим само в класа на нареденото ни дърво. Ето и неговата дефиниция:

```
...
/// <summary>
/// Represents a binary tree node
/// </summary>
/// <typeparam name="T"></typeparam>
private class BinaryTreeNode<T> : IComparable<BinaryTreeNode<T>>
  where T : IComparable<T>
{
  // Contains the value of the node
  internal T value;

  // Contains the parent of the node
  internal BinaryTreeNode<T> parent;

  // Contains the left child of the node
  internal BinaryTreeNode<T> leftChild;

  // Contains the right child of the node
  internal BinaryTreeNode<T> rightChild;


  /// <summary>
  /// Constructs the tree node
  /// </summary>
  /// <param name="value">The value of the tree node</param>
  public BinaryTreeNode(T value)
  {
    this.value = value;
    this.parent = null;
    this.leftChild = null;
    this.rightChild = null;
  }

  public override string ToString()
  {
    return this.value.ToString();
  }

  public override int GetHashCode()
  {
    return this.value.GetHashCode();
  }

  public override bool Equals(object obj)
  {
    BinaryTreeNode<T> other = (BinaryTreeNode<T>)obj;
    return this.CompareTo(other) == 0;
    }

    public int CompareTo(BinaryTreeNode<T> other)
    {
        return this.value.CompareTo(other.value);
    }
}
...
```

Да разгледаме предложения код. Още в името на структурата, която разглеждаме – "наредено дърво за търсене", ние говорим за наредба, а такава можем да постигнем **само** ако имаме **сравнимост** между елементите в дървото.

Сравнимост между обекти в C#

Какво означава понятието "сравнимост между обекти" за нас като програмисти? Това означава, че трябва да задължим по някакъв начин всички, които използват нашата структура от данни, да я създават подавайки и **тип, който е сравним**. На C# изречението "тип, който е сравним" би "звучало" така:

```
T : IComparable<T>
```

Интерфейсът **`IComparable<T>`**, намиращ се в пространството от имена **`System`**, се състои само от един метод `int` **`CompareTo(T obj)`**, който връща отрицателно цяло число, нула или положително цяло число съответно, ако текущият обект е по-малък, равен или по-голям от този, който е подаден на метода. Дефиницията му изглежда по приблизително следния начин:

```
public interface IComparable<T>
{
    // Summary:
    //     Compares the current object with another object of_the
    //     same type.
    int CompareTo(T other);
}
```

Имплементирането на този интерфейс от даден клас ни гарантира, че неговите инстанции са сравними.

От друга страна, на нас ни е необходимо и самите върхове, описани чрез класа **`BinaryTreeNode`**, също да бъдат сравними помежду си. Затова той също имплементира **`IComparable<T>`**. Както се вижда от кода, имплементацията на **`IComparable<T>`** на класа **`BinaryTreeNode`** вътрешно извиква тази на типа **`T`**.

В кода също сме припокрили и методите **`Equals(Object obj)`** и **`GetHashCode()`**. Добра (задължителна) практика е тези два метода да са съгласувани в поведението си т.е. когато два обекта са еднакви, хеш-кодът им да е еднакъв. Както ще видим в главата за хеш-таблици, обратното въобще не е задължително. Аналогично - очакваното поведение на **`Equals(Object obj)`** е да връща истина, точно когато и **`CompareTo(T obj)`** връща 0.

Задължително синхронизирайте работата на методите `Equals(Object obj)`, `CompareTo(T obj)` и `GetHashCode()`. Това е тяхното очаквано поведение и ще ви спести много трудно откриваеми проблеми!

До тук разгледахме методите, предложени от нашият клас. Сега да видим, какви полета ни предоставя. Те са съответно за **`value`** (ключът) от тип **`T`** родител – **`parent`**, ляв и десен наследник – съответно **`leftChild`** и **`rightChild`**. Последните три са от типа на дефиниращия ги клас, а именно **`BinaryTreeNode`**.

Наредени двоични дървета - реализация на основния клас

Преминаваме към реализацията на класа, описващ самото наредено двоично дърво. Дървото само по себе си като структура се състои от един корен от тип **`BinaryTreeNode`**, който вътрешно съдържа наследниците си – съответно ляв и десен, те вътрешно също съдържат техните наследници и така рекурсивно надолу докато се стигне до листата. Друго важно за отбелязване нещо е дефиницията **`BinarySearchTree<T> where T:IComparable<T>`**. Това ограничение на типа **`T`** се налага заради изискването на вътрешния ни клас, който работи само с типове, имплементиращи **`IComparable<T>`**.

```
public class BinarySearchTree<T>
   where T : IComparable<T>
{
   /// <summary>
   /// Represents a binary tree node
   /// </summary>
   /// <typeparam name="T">The type of the nodes</typeparam>
   private class BinaryTreeNode<T> :
      IComparable<BinaryTreeNode<T>>
      where T : IComparable<T>
   {
      //...
      //... The implementation from above goes here!!! ...
      //...
   }

   /// <summary>
   /// The root of the tree
   /// </summary>
   private BinaryTreeNode<T> root;
   /// <summary>
   /// Constructs the tree
   /// </summary>
   public BinarySearchTree()
   {
      this.root = null;
   }

   //...
   //... The operation implementation goes here!!! ...
   //...
}
```

Както споменахме по-горе, ще разгледаме следните операции:

- добавяне на елемент;
- търсене на елемент;
- изтриване на елемент.

Добавяне на елемент в подредено двоично дърво

След добавяне на нов елемент, дървото трябва да запази своята нареденост. Алгоритъмът е следният: ако дървото е празно, то добавяме новия елемент като корен. В противен случай:

- Ако елементът е по-малък от корена, то се обръщаме рекурсивно към същия метод, за да включим елемента в лявото поддърво.
- Ако елементът е по-голям от корена, то се обръщаме рекурсивно към същия метод, за да включим елемента в дясното поддърво.
- Ако елементът е равен на корена, то не правим нищо и излизаме от рекурсията.

Ясно се вижда как алгоритъмът за включване на връх изрично се съобразява с правилото елементите в лявото поддърво да са по-малки от корена на дървото и елементите от дясното поддърво да са по-големи от корена на дървото. Ето и примерна имплементация на този метод. Забележете, че при включването се поддържа референция към родителя, защото родителят също трябва да бъде променен.

```
/// <summary>
/// Inserts new value in the binary search tree
/// </summary>
/// <param name="value">the value to be inserted</param>
public void Insert(T value)
{
   if (value == null)
   {
      throw new ArgumentNullException(
            "Cannot insert null value!");
    }

    this.root = Insert(value, null, root);
}

/// <summary>
/// Inserts node in the binary search tree by given value
/// </summary>
/// <param name="value">the new value</param>
/// <param name="parentNode">the parent of the new node</param>
/// <param name="node">current node</param>
/// <returns>the inserted node</returns>
private BinaryTreeNode<T> Insert(T value,
        BinaryTreeNode<T> parentNode,
        BinaryTreeNode<T> node)
{
    if (node == null)
    {
        node = new BinaryTreeNode<T>(value);
        node.parent = parentNode;
    }
    else
    {
        int compareTo = value.CompareTo(node.value);
        if (compareTo < 0)
        {
            node.leftChild =
                Insert(value, node, node.leftChild);
        }
        else if (compareTo > 0)
        {
            node.rightChild =
                Insert(value, node, node.rightChild);
        }
    }

    return node;
}
```

Търсене на елемент в подредено двоично дърво

Търсенето е операция, която е още по-интуитивна. В примерния код сме показали как може търсенето да се извърши без рекурсия, а чрез итерация. Алгоритъмът започва с елемент **node**, сочещ корена. След това се прави следното:

- Ако елементът е равен на **node**, то сме намерили търсения елемент и го връщаме.

- Ако елементът е по-малък от **`node`**, то присвояваме на **`node`** левия му наследник т.е. продължаваме търсенето в лявото поддърво.
- Ако елементът е по-голям от **`node`**, то присвояваме на **`node`** десния му наследник т.е. продължаваме търсенето в дясното поддърво.

При приключване, алгоритъмт връща или намерения връх или **`null`**, ако такъв връх не съществува в дървото.

Следва примерен код:

```
/// <summary>
/// Finds a given value in the tree and returns the node
/// which contains it if such exsists
/// </summary>
/// <param name="value">the value to be found</param>
/// <returns>the found node or null if not found</returns>
private BinaryTreeNode<T> Find(T value)
{
   BinaryTreeNode<T> node = this.root;

   while (node != null)
   {
      int compareTo = value.CompareTo(node.value);
      if (compareTo < 0)
      {
         node = node.leftChild;
      }
      else if (compareTo > 0)
      {
         node = node.rightChild;
      }
      else
      {
         break;
      }
   }

   return node;
}
```

Изтриване на елемент от подредено двоично дърво

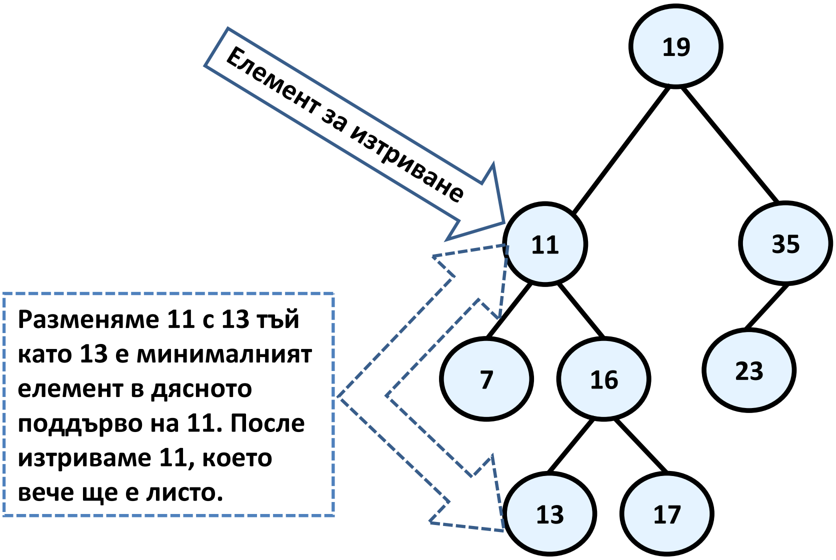
Изтриването е най-сложната операция от трите основни. След нея дървото трябва да запази своята нареденост. Първата стъпка преди да изтрием елемент от дървото е да го намерим. Вече знаем как става това. След това се прави следното:

- Ако върхът е листо – насочваме референцията на родителя му към **`null`**. Ако елементът няма родител следва, че той е корен и просто го изтриваме.

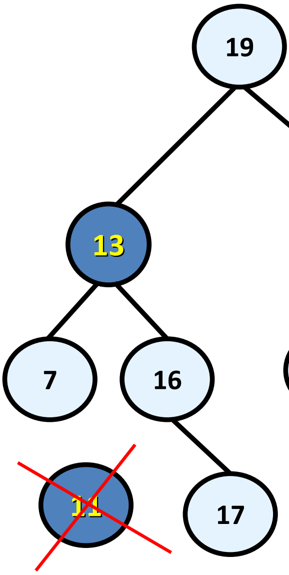
- Ако върхът има само едно поддърво – ляво или дясно, то той се замества с корена на това поддърво.
- Ако върхът има две поддървета. Тогава намираме най-малкият връх в дясното му поддърво и го разменяме с него. След тази размяна върхът ще има вече най-много едно поддърво и го изтриваме по някое от горните две правила. Тук трябва да отбележим, че може да се направи аналогична размяна, само че взимаме лявото поддърво и най-големият елемент от него.

Оставяме на читателя, като леко упражнение, да провери коректността на всяка една от тези три стъпки.

Нека разгледаме едно примерно изтриване. Ще използваме отново нашето наредено дърво, което показахме в началото на тази точка. Да изтрием например елемента с ключ 11.


Елемент за изтриване
19
11
35
7
16
23
13
17
Разменяме 11 с 13 тъй като 13 е минималният елемент в дясното поддърво на 11. После изтриваме 11, което вече ще е листо.

Върхът 11 има две поддървета и, съгласно нашия алгоритъм, трябва да бъде разменен с най-малкия елемент от дясното поддърво, т.е. с 13. След като извършим размяната вече можем спокойно да изтрием 11, който е листо. Ето крайния резултат:



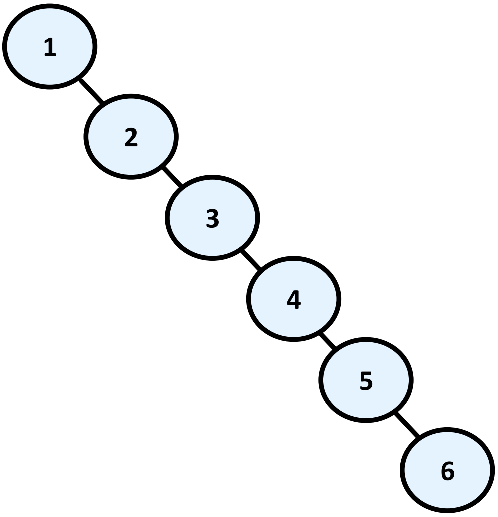
Предлагаме следния примерен код, който реализира описания алгоритъм:

```
/// <summary>
/// Removes an element from the tree if exists
/// </summary>
/// <param name="value">the value to be deleted</param>
public void Remove(T value)
{
   BinaryTreeNode<T> nodeToDelete = Find(value);
   if (nodeToDelete == null)
   {
      return;
   }

   Remove(nodeToDelete);
}

private void Remove(BinaryTreeNode<T> node)
{
   // Case 3: If the node has two children.
   // Note that if we get here at the end
   // the node will be with at most one child
   if (node.leftChild != null && node.rightChild != null)
   {
      BinaryTreeNode<T> replacement = node.rightChild;
      while (replacement.leftChild != null)
      {
         replacement = replacement.leftChild;
      }
      node.value = replacement.value;
      node = replacement;
```

```
    }

    // Case 1 and 2: If the node has at most one child
    BinaryTreeNode<T> theChild = node.leftChild != null ?
            node.leftChild : node.rightChild;

    // If the element to be deleted has one child
    if (theChild != null)
    {
        theChild.parent = node.parent;

        // Handle the case when the element is the root
        if (node.parent == null)
        {
            root = theChild;
        }
        else
        {
            // Replace the element with its child subtree
            if (node.parent.leftChild == node)
            {
                node.parent.leftChild = theChild;
            }
            else
            {
                node.parent.rightChild = theChild;
            }
        }
    }
    else
    {
        // Handle the case when the element is the root
        if (node.parent == null)
        {
            root = null;
        }
        else
        {
            // Remove the element - it is a leaf
            if (node.parent.leftChild == node)
            {
                node.parent.leftChild = null;
            }
            else
            {
                node.parent.rightChild = null;
            }
        }
    }
}
```

Балансирани дървета

Както видяхме по-горе, наредените двоични дървета представляват една много удобна структура за търсене. Така дефинирани операциите за създаване и изтриване на дървото имат един скрит недостатък. Какво би станало ако в дървото включим последователно елементите 1, 2, 3, 4, 5, 6? Ще се получи следното дърво:



В този случай двоичното дърво се е изродило в свързан списък. От там и търсенето в това дърво ще е доста по-бавно (с **N** на брой стъпки, а не с **log(N)**), тъй като, за да проверим дали даден елемент е вътре, в най-лошия случай ще трябва да преминем през всички елементи.

Ще споменем накратко за съществуването на структури от данни, които в общия случай запазват логаритмичното поведение на операциите добавяне, търсене и изтриване на елемент. Преди да кажем как се постига това, ще въведем следните две дефиниции:

Балансирано двоично дърво – двоично дърво, в което никое листо не е на "много по-голяма" дълбочина от всяко друго листо. Дефиницията на "много по-голяма" зависи от конкретната балансираща схема.

Идеално балансирано двоично дърво – двоично дърво, в което разликата в **броя на върховете на лявото и дясното поддърво** на всеки от върховете е най-много единица.

Без да навлизаме в детайли ще споменем, че ако дадено двоично дърво е балансирано, дори и да не е идеално балансирано, то операциите за добавяне, търсене и изтриване на елемент в него са с логаритмична сложност дори и в най-лошия случай. За да се избегне дисбаланса на дървото за претърсване, се прилагат операции, които пренареждат част от елементите на дървото при добавяне или при премахване на елемент от него. Тези операции най-често се наричат **ротации**. Конкретният вид на ротациите, се уточнява допълнително и зависи от реализацията на конкретната структура от данни. Като примери за такива структури, можем да дадем **червено-черно** дърво, **AVL**-дърво, **AA**-дърво, **Splay**-дърво и др.

За по-детайлно разглеждане на тези и други структури препоръчваме на читателя да потърси в строго специализираната литература за алгоритми и структури от данни.

Скритият клас TreeSet<T> в .NET Framework

След като вече се запознахме с наредените двоични дървета и с това какво е предимството те да са балансирани, идва момента да покажем и какво C# има за нас по този въпрос. Може би всеки от вас тайно се е надявал, че никога няма да му се налага да имплементира балансирано наредено двоично дърво за търсене, защото изглежда доста сложно. Това най-вероятно наистина е така.

До момента разгледахме какво представляват балансираните дървета, за да добиете представа за тях. Когато ви се наложи да ги ползвате, винаги можете да разчитате да ги вземете от някъде наготово. В стандартните библиотеки на .NET Framework има готови имплементации на балансирани дървета, а освен това по Интернет можете да намерите и много външни библиотеки.

В пространството от имена **`System.Collections.Generic`** се поддържа класът **`TreeSet<T>`**, който вътрешно представлява имплементация на червено-черно дърво. Това, както вече знаем, означава, че добавянето, търсенето и изтриването на елементи в дървото ще се извърши с логаритмична сложност (т.е. ако имаме 1 000 000 елемента операцията ще бъде извършена за около 20 стъпки). Лошата новина е, че този клас е **`internal`** и е видим само в тази библиотека. За щастие обаче, този клас се ползва вътрешно от друг, който е публично достъпен – **`SortedDictionary<T>`**. Повече информация за класа **`SortedDictionary<T>`** можете да намерите в секцията "Множества" на главата "Речници, хеш-таблици и множества".

Графи

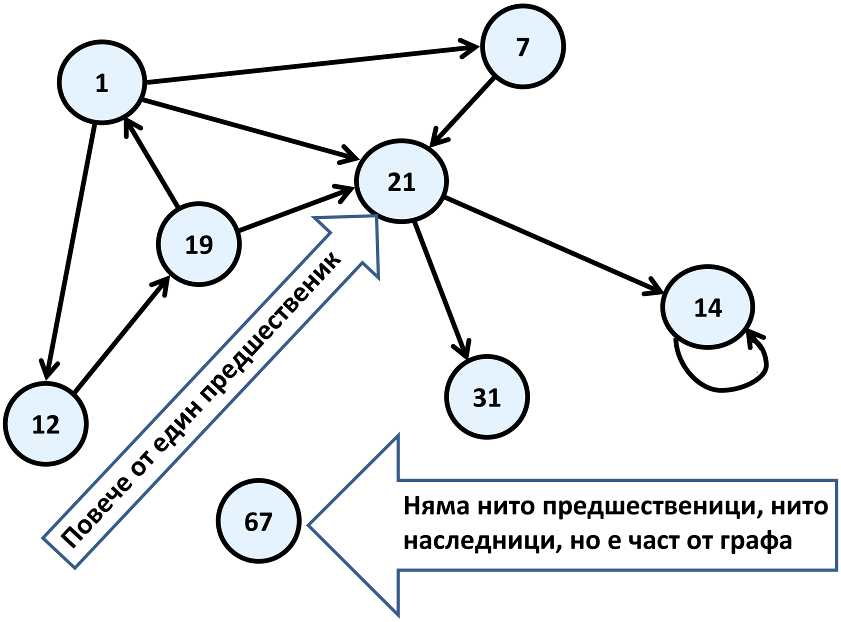
Графите се една изключително полезна и доста разпространена структура от данни. Използват се за описването на най-разнообразни взаимовръзки между обекти от практиката, свързани с почти всичко. Както ще видим по-късно, дървета са подмножество на графите, т.е. графите представляват една обобщена структура, позволяваща моделирането на доста голяма съвкупност от реални ситуации.

Честата употреба на графите в практиката е довела до задълбочени изследвания в "теория на графите", в която са известни огромен брой задачи за графи и за повечето от тях има и добре известно решение.

Графи – основни понятия

В тази точка ще въведем някои от по-важните понятия и дефиниции. Част от тях са аналогични на тези, въведени при структурата от данни дърво, но двете структури, както ще видим, имат много сериозни различия, тъй като дървото е само един частен случай на граф.

Да разгледаме следният примерен граф, чийто тип по-късно ще наречем краен ориентиран. В него отново имаме номерация на върховете, която е абсолютно произволна и е добавена, за да може по-лесно да говорим за някой от тях конкретно:


1
7
21
19
14
31
12
67
Повече от един предшественик
Няма нито предшественици, нито наследници, но е част от графа

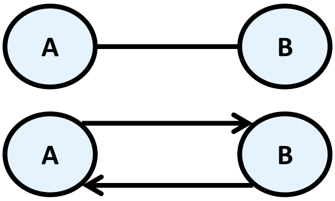
Кръгчетата на схемата, ще наричаме **върхове**, а стрелките, които ги свързват, ще наричаме **ориентирани ребра (дъги)**. Върхът, от който излиза стрелката ще наричаме **предшественик** на този, който стрелката сочи. Например "19" е предшественик на "1". "1" от своя страна се явява **наследник** на "19". За разлика от структурата дърво, сега всеки един връх може да има повече от един предшественик. Например "21" има три - "19", "1" и "7". Ако два върха са свързани с ребро, то казваме, че тези два върха са **инцидентни** с това ребро.

Следва дефиниция за **краен ориентиран граф (finite directed graph)**:

Краен ориентиран граф се нарича наредената двойката двойка **(V, E)**, където **V** е крайно множество от върхове, а **E** е крайно множество от ориентирани ребра. Всяко ребро ***e*** принадлежащо на **E** представлява наредена двойка от върхове ***u*** и ***v*** т.е. ***e*=(*u*, *v*)**, които еднозначно го определят.

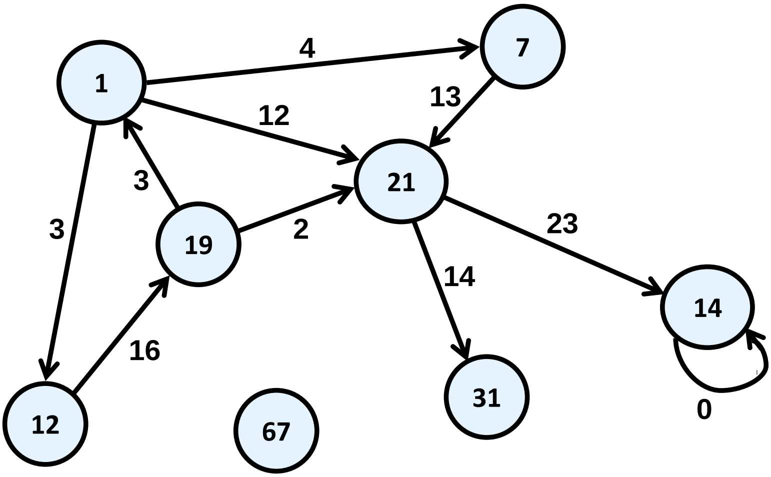
За по-доброто разбиране на тази дефиниция силно препоръчваме на читателя да си мисли за върховете например като за градове, а ориентираните ребра като еднопосочни пътища. Така, ако единият връх е София, а другият е Велико Търново, то еднопосочният път (дъгата) ще се нарича София-Велико Търново. Всъщност това е един от класическите примери за приложение на графите – в задачи свързани с пътища.

Ако вместо със стрелки върховете са свързани с отсечки, то тогава отсечките ще наричаме **неориентирани ребра**, а графът – **неориентиран**. На практика можем да си представяме, че едно неориентирано ребро от връх A до връх B представлява двупосочно ребро, еквивалентно на две противоположни ориентирани ребра между същите два върха:



Два върха, свързани с ребро, ще наричаме **съседни**.

За ребрата може да се зададе функция, която на всяко едно ребро съпоставя реално число. Тези така получени реални числа ще наричаме **тегла**. Като примери за тегла можем да дадем дължината на директните връзки между два съседни града, пропускателната способност на една тръба и др. Граф, който има тегла по ребрата, се нарича **претеглен (weighted)**. Ето как се изобразява претеглен граф:



Път в граф ще наричаме последователност от върхове v_1, v_2, … , v_n, такава, че съществува ребро от v_i до v_{i+1} за всяко i от 1 до n-1. В нашия граф път е например последователността "1", "12", "19", "21". "7", "21" и "1" обаче не е път, тъй като не съществува ребро започващо от "21" и завършващо в "1".

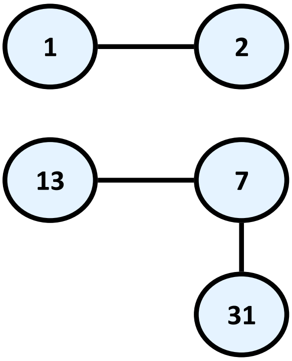
Дължина на път е броят на ребрата, свързващи последователността от върхове в пътя. Този брой е равен на броя на върховете в пътя минус единица. Дължината на примера ни за път "1", "12", "19", "21" е три.

Цена на път в претеглен граф, ще наричаме сумата от теглата на ребрата участващи в пътя. В реалния живот пътят от София до Варна, например, е равен на дължината на пътя от София до Велико Търново плюс дължината на пътя от Велико Търново до Варна. В нашия пример дължината на пътя "1", "12", "19" и "21" е равна на 3 + 16 + 2 = 21.

Цикъл е път, в който началният и крайният връх на пътя съвпадат. Пример за върхове, образуващи цикъл, са "1", "12" и "19". "1", "7" и "21" обаче не образуват цикъл.

Примка ще наричаме ребро, което започва от и свършва в един и същ връх. В нашия пример върха "14" има примка.

Свързан неориентиран граф наричаме неориентиран граф, в който съществува път от всеки един връх до всеки друг. Например следният граф не е свързан, защото не съществува път от "1" до "7".



И така, вече имаме достатъчно познания, за да дефинираме понятието дърво по още един начин – като специален вид граф:

Дърво – неориентиран свързан граф без цикли.

Като леко упражнение оставяме на читателя да покаже защо двете дефиниции за дърво са еквивалентни.

Графи – видове представяния

Съществуват много различни начини за представяне на граф в програмирането. Различните представяния имат различни свойства и кое точно трябва да бъде избрано, зависи от конкретния алгоритъм, който искаме да приложим. С други думи казано – представяме графа си така, че операциите, които алгоритъмът ни най-често извършва върху него, да бъдат максимално бързи. Без да изпадаме в големи детайли ще изложим някои от най-често срещаните представяния на графи.

- **Списък на ребрата** – представя се, чрез списък от наредени двойки (v_i, v_j), където съществува ребро от $\mathbf{v_i}$ до $\mathbf{v_j}$. Ако графът е претеглен, то вместо наредена двойка имаме наредена тройка, като третият ѝ елемент показва какво е теглото на даденото ребро.

- **Списък на наследниците** – в това представяне за всеки връх **v** се пази списък с върховете, към които сочат ребрата започващи от **v**. Тук отново, ако графът е претеглен, към всеки елемент от списъка с наследниците се добавя допълнително поле, показващо цената на реброто до него.
- **Матрица на съседство** – графът се представя като квадратна матрица `g[N][N]`, в която, ако съществува ребро от $\mathbf{v}_i$ до $\mathbf{v}_j$, то на позиция `g[i][j]` в матрицата е записано 1. Ако такова ребро не съществува, то в полето `g[i][j]` е записано 0. Ако графът е претеглен, в позиция `g[i][j]` се записва теглото на даденото ребро, а матрицата се нарича **матрица на теглата.** Ако между два върха в такава матрица не съществува път, то тогава се записва специална стойност, означаваща безкрайност.
- **Матрица на инцидентност между върхове и ребра –** в този случай отново се използва матрица, само че с размери `g[M][N]`, където M е броят на върховете, а N е броят на ребрата. Всеки стълб представя едно ребро, а всеки ред един връх. Тогава в стълба съответстващ на реброто ($\mathbf{v}_i$, $\mathbf{v}_j$) само и единствено на позиция **i** и на позиция **j** ще бъдат записани 1, а на останалите позиции в този стълб ще е записана 0. Ако реброто е примка, т.е. е ($\mathbf{v}_i$, $\mathbf{v}_i$), то на позиция **i** записваме 2. Ако графът, който искаме да представим, е ориентиран и искаме да представим ребро от $\mathbf{v}_i$ до $\mathbf{v}_j$, то на позиция **i** пишем 1, а на позиция **j** пишем -1.

Графи – основни операции

Основните операции в граф са:

- Създаване на граф
- Добавяне / премахване на връх / ребро
- Проверка дали даден връх / ребро съществува
- Намиране на наследниците на даден връх

Ще предложим примерна реализация на представяне на граф с матрица на съседство и ще покажем как се извършват повечето операции. Този вид реализация е удобен, когато максималният брой на върховете е предварително известен и когато той не е много голям (за да се реализира представянето на граф с N върха е необходима памет от порядъка на N^2 заради квадратната матрица). Поради това, няма да реализираме методи за добавяне / премахване на нов връх.

```
using System;
using System.Collections.Generic;

```

```
/// <summary>
/// Represents a directed unweighted graph structure.
/// </summary>
public class Graph
{
   // Contains the vertices of the graph
   private int[,] vertices;

   /// <summary>
   /// Constructs the graph.
   /// </summary>
   /// <param name="vertices">the vertices of the graph</param>
   public Graph(int[,] vertices)
   {
      this.vertices = vertices;
   }

   /// <summary>
   /// Adds new edge from i to j.
   /// </summary>
   /// <param name="i">the starting vertex</param>
   /// <param name="j">the ending vertex</param>
   public void AddEdge(int i, int j)
   {
      vertices[i,j] = 1;
   }

   /// <summary>
   /// Removes the edge from i to j if such exists.
   /// </summary>
   /// <param name="i">the starting vertex</param>
   /// <param name="j">the ending vertex</param>
   public void RemoveEdge(int i, int j)
   {
      vertices[i,j] = 0;
   }

   /// <summary>
   /// Checks whether there is an edge between vertex i and j.
   /// </summary>
   /// <param name="i">the starting vertex</param>
   /// <param name="j">the ending vertex</param>
   /// <returns>true if there is an edge between
   /// vertex i and vertex j</returns>
   public bool HasEdge(int i, int j)
   {
      return vertices[i,j] == 1;
   }

   /// <summary>
   /// Returns the successors of a given vertex.
   /// </summary>
   /// <param name="i">the vertex</param>
   /// <returns>list with all successors of the given
vertex</returns>
   public IList<int> GetSuccessors(int i)
   {
      IList<int> successors = new List<int>();

      for (int j = 0; j < vertices.GetLength(1); j++)
      {
         if (vertices[i,j] == 1)
         {
            successors.Add(j);
         }
      }

      return successors;
   }
}
```

Основни приложения и задачи за графи

Графите се използват за моделиране на много ситуации от реалността, а задачите върху графи моделират множество реални проблеми, които често се налага да бъдат решавани. Ще дадем само няколко примера:

- **Карта на град** може да се моделира с ориентиран претеглен граф. На всяка улица се съпоставя ребро с дължина съответстваща на дължината на улицата и посока – посоката на движение. Ако улицата е двупосочна може да й се съпоставят две ребра за двете посоки на движение. На всяко кръстовище се съпоставя връх. При такъв модел са естествени задачи като търсене на най-кратък път между две кръстовища, проверка дали има път между две кръстовища, проверка за цикъл (дали можем да се завъртим и да се върнем на изходна позиция), търсене на път с минимален брой завои и т.н.

- **Компютърна мрежа** може да се моделира с неориентиран граф, чиито върхове съответстват на компютрите в мрежата, а ребрата съответстват на комуникационните канали между компютрите. На ребрата могат да се съпоставят различни числа, примерно капацитет на канала или скорост на обмена и др. Типични задачи при такива модели на компютърна мрежа са проверка за свързаност между два компютъра, проверка за двусвързаност между две точки (съществуване на двойно-подсигурен канал, който остава при отказ на който и да е компютър) и др. В частност Интернет може да се моделира като граф, в който се решават задачи за маршрутизация на пакети, които се моделират като задачи за графи.

- **Речната система** в даден регион може да се моделира с насочен претеглен граф, в който всяка река се състои от едно или няколко ребра, а всеки връх съответства на място, където две или повече реки се вливат една в друга. По ребрата могат да се съпоставят стойности, свързани с количеството вода, което преминава по тях. Естествени при този модел са задачи като изчисление на обемите вода, преминаващи през всеки връх и предвиждане на евентуални наводнения при увеличаване на количествата.

Виждате, че графите могат да имат многобройни приложения. За тях има изписани стотици книги и научни трудове. Съществуват десетки класически задачи за графи, за които има известни решения или е известно, че нямат ефективно решение. Ние няма да се спираме на тях. Надяваме се чрез краткото представяне да събудим интереса ви към темата и да ви подтикнем да отделите достатъчно внимание на задачите за графи от упражненията.

Упражнения

1. Да се напише програма, която намира броя на срещанията на дадено число в дадено дърво от числа.
2. Да се напише програма, която извежда корените на онези поддървета на дадено дърво, които имат точно **k** на брой върха, където **k** е дадено естествено число.
3. Да се напише програма, която намира броя на листата и броя на вътрешните върхове на дадено дърво.
4. Напишете програма, която по дадено двоично дърво от числа намира сумата на върховете от всяко едно ниво на дървото.
5. Да се напише програма, която намира и отпечатва всички върхове на двоично дърво, които имат за наследници само листа.
6. Да се напише програма, която проверява дали дадено двоично дърво е идеално балансирано.
7. Нека е даден граф **G(V, E)** и два негови върха **x** и **y**. Напишете програма, която намира най-краткия път между два върха по брой на върховете.
8. Нека е даден граф **G(V, E)**. Напишете програма, която проверява дали графът е цикличен.
9. Напишете рекурсивно обхождане в дълбочина и програма, която да го тества.
10. Напишете обхождане в ширина (BFS), базирано на опашка.
11. Напишете програма, която обхожда директорията C:\Windows\ и всичките и поддиректории рекурсивно и отпечатва всички файлове, който имат разширение *.exe.

12. Дефинирайте класове File { string name, int size } и Folder { string name, File[] files, Folder[] childFolders }. Използвайки тези класове, постройте дърво, което съдържа всички файлове и директории на твърдия диск, като започнете от C:\Windows\. Напишете метод, който изчислява сумата от големините на файловете в дадено поддърво и програма, която тества този метод. За обхождането на директориите използвайте рекурсивно обхождане в дълбочина (DFS).
13. Напишете програма, която намира всички цикли в даден граф.
14. Нека е даден граф **G(V, E)**. Напишете програма, която намира всички компоненти на свързаност на графа, т.е. намира всички негови максимални свързани подграфи. Максимален свързан подграф на **G** е свързан граф такъв, че няма друг подграф на **G**, който да е свързан и да го съдържа.
15. Нека е даден претеглен ориентиран граф **G(V, E)**, в който теглата по ребрата са неотрицателни числа. Напишете програма, която по зададен връх **x** от графа намира минималните пътища от него до всички останали.
16. Имаме N задачи, които трябва да бъдат изпълнени последователно. Даден е списък с двойки задачи, за които втората зависи от резултата от първата и трябва да бъде изпълнена след нея. Напишете програма, която подрежда задачите по такъв начин, че всяка задача да се изпълни след всички задачи, от които зависи. Ако не съществува такава наредба, да се отпечата подходящо съобщение.

    Пример: `{1, 2}, {2, 5}, {2, 4}, {3, 1}` → `3, 1, 2, 5, 4`
17. Ойлеров цикъл в граф се нарича цикъл, който започва от даден връх, минава точно по веднъж през всички негови ребра и се връща в началния връх. При това обхождане всеки връх може да бъде посетен многократно. Напишете програма, която по даден граф намира в него Ойлеров цикъл или установява, че такъв няма.
18. Хамилтонов цикъл в граф се нарича цикъл, съдържащ всеки връх в графа точно по веднъж. Да се напише програма, която при даден претеглен ориентиран граф **G(V, E)**, намира Хамилтонов цикъл с минимална дължина, ако такъв съществува.

Решения и упътвания

1. Обходете рекурсивно дървото в дълбочина и пребройте срещанията на даденото число.
2. Обходете рекурсивно дървото в дълбочина и проверете за всеки връх даденото условие.
3. Можете да решите задачата с рекурсивно обхождане на дървото в дълбочина.

4. Използвайте обхождане в дълбочина или в ширина и при преминаване от един връх в друг запазвайте в него на кое ниво се намира. Знаейки нивата на върховете търсената сума лесно се изчислява.

5. Можете да решите задачата с рекурсивно обхождане на дървото в дълбочина и проверка на даденото условие.

6. Чрез рекурсивно спускане в дълбочина за всеки връх на дървото изчислете дълбочините на лявото и дясното му поддърво. След това проверете непосредствено дали е изпълнено условието от дефиницията за идеално балансирано дърво.

7. Използвайте като основа алгоритъма за обхождане в ширина. Слагайте в опашката заедно с даден връх и неговия предшественик. Това ще ви помогне накрая да възстановите пътя между върховете (в обратен ред).

8. Използвайте обхождане в дълбочина или в ширина. Отбелязвайте за всеки връх дали вече е бил посетен. Ако в даден момент достигнете до връх, който е бил посетен по-рано, значи сте намерили цикъл.

   Помислете как можете да намерите и отпечатате самия цикъл. Ето една възможна идея: при обхождане в дълбочина за всеки връх пазите предшественика му. Ако в даден момент стигнете до връх, който вече е бил посетен, вие би трябвало да имате запазен за него някакъв път до началния връх. Текущият път в стека на рекурсията също е път до въпросния връх. Така в даден момент имаме два различни пътя от един връх до началния връх. От двата пътя лесно можете да намерите цикъл.

9. Използвайте алгоритъма DFS.

10. Използвайте **`Queue<Vertex>`**

11. Използвайте обхождане в дълбочина и класа **`System.IO.Directory`**.

12. Използвайте примера за дърво по горе. Всяка директория от дървото има наследници поддиректориите, и стойност файловете в

13. Използвайте задача 8, но я променете да не спира когато намери един цикъл а да продължава. За всеки цикъл трябва да проверите дали вече не сте го намерили

14. Използвайте като основа алгоритъма за обхождане в ширина или в дълбочина.

15. Използвайте алгоритъма на Dijkstra (намерете го в Интернет).

    Търсената наредба се нарича "топологично сортиране на ориентиран граф". Може да се реализира по два начина:

    За всяка задача **`t`** пазим от колко на брой други задачи **`P(t)`** зависи. Намираме задача t_0, която не зависи от никоя друга ($P(t_0)=0$) и я изпълняваме. Намаляваме P(t) за всяка задача t, която зависи от t_0.

Отново търсим задача, която не зависи от никоя друга и я изпълняваме. Повтаряме докато задачите свършат или до момент, в който няма нито една задача t_k с $P(t_k)=0$.

16. Можем да решим задачата чрез обхождане в дълбочина на графа и печатане на всеки връх при напускането му. Това означава, че в момента на отпечатването на дадена задача всички задачи, които зависят от нея са били вече отпечатани.

17. За да съществува Ойлеров цикъл в даден граф, трябва графът да е свързан и степента на всеки негов връх да е четно число. Чрез поредица впускания в дълбочина можете да намирате цикли в графа и да премахвате ребрата, които участват в тях. Накрая като съедините циклите един с друг ще получите Ойлеров цикъл.

18. Ако напишете вярно решение на задачата, проверете дали работи за граф с 200 върха. Не се опитвайте да решите задачата, така че да работи бързо за голям брой върхове. Ако някой успее да я реши, ще остане трайно в историята!

Telerik Academy SCHOOL

Стани част от Академията на Телерик за ученици!

“Основната цел на обучението е да израстнеш, а нашите умове, за разлика от телата ни, могат да растат през целия ни живот!“ Мортимър Адлър

АКАДЕМИЯТА НА ТЕЛЕРИК за ученици предоставя съвременно, специализирано обучение по софтуерно инженерство за ученици от средните училища в България. Нашата основна цел е да помогнем на младите момчета и момичета да се подготвят за Националната олимпиада по информационни технологии (НОИТ) и да придобият солидни теоретични и практически умения за разработка на софтуер и софтуерни технологии.

В Училищната академия на Телерик по софтуерно инженерство:

- **Стимулираме креативността и мисленето на учениците**
- **Всеки ученик получава индивидуално внимание и съдействие**
- **Полагаме основите за писане на качествен програмен код**
- **Изучаваме съвременните софтуерни технологии**
- **Подготвяме учениците за Националната олимпиада по информационни технологии**
- **Предлагаме голямо разнообразие от практически и теоретични обучения**

Програмата на Академията включва:

C#, .NET технологии, бази данни, SQL, Silverlight, 3D графики и моделиране, разработка на игри, програмиране на микроконтролери, WPF, ASP.NET, HTML5, разработка на мобилни приложения, основи на софтуерното инженерство, работа в екип

schoolacademy.telerik.com

groups.google.com/group/it-olymp   *facebook.com/TelerikSchoolAcademy*

telerik
deliver more than expected

Глава 18. Речници, хеш-таблици и множества

В тази тема...

В настоящата тема ще разгледаме някои по-сложни структури от данни като речници и множества, и техните реализации с хеш-таблици и балансирани дървета. Ще обясним в детайли какво представляват хеширането и хеш-таблиците и защо са токова важни в програмирането. Ще дискутираме понятието "колизия, как се получават колизиите при реализация на хеш-таблици и ще предложим различни подходи за разрешаването им. Ще разгледаме абстрактната структура данни "множество" и ще обясним как може да се реализира чрез речник и чрез балансирано дърво. Ще дадем примери, които илюстрират приложението на описаните структури от данни в практиката.

Структура от данни "речник"

В предните няколко теми се запознахме с някои класически и много важни структури от данни – масиви, списъци, дървета и графи. В тази - ще се запознаем с така наречените "**речници**" (**dictionaries**), които са изключително полезни и широко използвани в програмирането.

Речниците са известни още като **асоциативни масиви (associative arrays)** или **карти (maps)**. Тук ще използваме терминът "речник". Всяко едно от различните имена подчертава една и съща характеристика на тази структура от данни, а именно, че в тях всеки елемент представлява съответствие между ключ и стойност – наредена двойка. Аналогията идва от факта, че в един речник, например тълковния речник, за всяка дума (**ключ**) имаме обяснение (**стойност**). Подобни са тълкованията и на другите имена.

При речниците заедно с данните, които държим, пазим и ключ, по който ги намираме. Елементите на речниците са двойки (ключ, стойност), като ключът се използва при търсене.

Структура от данни "речник" – пример

Ще илюстрираме какво точно представлява структура от данни речник с един конкретен пример от ежедневието.

Когато отидете на театър, опера или концерт често преди да влезете в залата или стадиона има гардероб, в който може да оставите дрехите си. Там давате дрехата си на служителката от гардероба, тя я оставя на определено място и ви дава номерче. След като свърши представлението, на излизане давате вашето номерче, и чрез него служителката намира точно вашата дреха и ви я връща.

Чрез този пример виждаме, че идеята да разполагаме с ключ (номерче, което ви дава служителката) за данните (вашата дреха) и да ги достъпваме чрез него, не е толкова абстрактна. В действителност това е подход, който се среща на много места, както в програмирането така и в много сфери на реалния живот.

При структурата речник ключът може да не е просто номерче, а всякакъв друг обект. В случая, когато имаме ключ (номер), можем да реализираме такава структура като обикновен масив. Тогава множеството от ключове е предварително ясно – числата от `0` до `n`, където `n` е размерът на масива (естествено при разумно ограничение на `n`). Целта на речниците е да ни освободи, до колкото е възможно, от ограниченията за множеството на ключовете.

При речниците обикновено множеството от ключове е произволно множество от стойности, примерно реални числа или символни низове. Единственото задължително изискване е да можем да различим един ключ от друг. След малко ще се спрем по-конкретно на някои допълнителни изисквания към ключовете, необходими за различните реализации.

Речниците съпоставят на даден ключ дадена стойност. На един ключ може да се съпостави точно една стойност. Съвкупността от всички двойки (ключ, стойност) съставя речника.

Ето и първия пример за ползване на речник в .NET:

```
IDictionary<string, double> studentMarks =
    new Dictionary<string, double>();

studentMarks["Pesho"] = 3.00;
Console.WriteLine("Pesho's mark: {0:0.00}",
    studentMarks["Pesho"]);
```

По-нататък в главата ще разберем какъв ще бъде резултата от изпълнението на този пример.

Абстрактна структура данни "речник" (асоциативен масив, карта)

В програмирането абстрактната структура данни "речник" представлява съвкупност от наредени двойки (ключ, стойност), заедно с дефинирани операции за достъп до стойностите по ключ. Алтернативно тази структура може да бъде наречена още "карта" (map) или "асоциативен масив" (associative array).

Задължителни операции, които тази структура дефинира, са следните:

- **`void Add(K key, V value)`** – добавя в речника зададената наредена двойка. При повечето имплементации на класа в .NET, при добавяне на ключ, който вече съществува в речника, се хвърля изключение.
- **`V Get(K key)`** – връща стойността по даден ключ. Ако в речника няма двойка с такъв ключ, метода връща **`null`**, или хвърля изключение, според конкретната имплементация на речника
- **`bool Remove(key)`** – премахва стойността за този ключ от речника. Освен това връща дали е премахнат елемент от речника.

Ето и някои операции, които различните реализации на речници често предлагат:

- **`bool Contains(key)`** – връща **`true`**, ако в речникът има двойка с дадения ключ.
- **`int Count`** – връща броя елементи в речника.

Други операции, които обикновено се предлагат, са извличане на всички ключове, стойности или наредени двойки ключ-стойност, в друга структура (масив, списък). Така те лесно могат да бъдат обходени чрез цикъл.

За улеснение на .NET разработчиците, в интерфейса `IDictionary<K, V>` е добавено индексно свойство `V this[K] { get; set; }`, което обикновено се имплементира, чрез извикване на методите съответно `V Get(K)`, `Add(K, V)`. Трябва да имаме предвид, че метода за достъп (accessor) `get` на свойството `V this[K]` на класът `Dictionary<K, V>` в .NET хвърля изключение, ако даденият ключ K не присъства в речника. За да вземем стойността за даден ключ без да се опасяваме от изключения, можем да използваме метода - `bool TryGetValue(K key, out V value)`

Интерфейсът IDictionary<K, V>

В .NET има дефиниран стандартен интерфейс **`IDictionary<K, V>`** където **`K`** дефинира типа на ключа (**`key`**), а **`V`** типа на стойността (**`value`**). Той дефинира всички основни операции, които речниците трябва да реализират. **`IDictionary<K, V>`** съответства на абстрактната структура от данни "речник" и дефинира операциите, изброени по-горе, но без да предоставя конкретна реализация за всяка от тях. Този интерфейс е дефиниран в асембли **`mscorelib`**, **`namespace System.Collections.Generic`**.

В .NET интерфейсите представляват спецификации за методите на даден клас. Те дефинират методи без имплементация, които след това трябва да бъдат имплементирани от класовете, обявили, че поддържа дадения интерфейс. Как работят интерфейсите и наследяването ще разгледаме подробно в главата "Принципи на обектно-ориентираното програмиране". За момента е достатъчно да знаете, че интерфейсите задават какви методи и свойства трябва да имат всички класове, които го имплементират.

В настоящата тема ще разгледаме двата най-разпространени начина за реализация на речници – балансирано дърво и хеш-таблица. Изключително важно е да се знае, по какво се различават те един от друг и какви са основните принципи, свързани с тях. В противен случай рискувате да ги използвате неправилно и неефективно.

В .NET има две основни имплементации на интерфейса **`IDictionary<K, V>`**: **`Dictionary<K, V>`** и **`SortedDictionary<K, V>`**. **`SortedDictionary`** представлява имплементация с балансирано (червено-черно) дърво, а **`Dictionary`** – имплементация с хеш-таблица.

Освен `IDictionary<K, V>` в .NET има още един интерфейс - `IDictionary`, както и класове които го имплементират: `Hashtable`, `ListDictionary`, `HybridDictionary`. Те са наследство от първите версии на .NET. Тези класове трябва да се ползват само при специфична нужда. За предпочитане е употребата на `Dictionary<K, V>` или `SortedDictionary<K, V>`.

В тази и следващата тема ще разгледаме в кои случаи се използват различните имплементации на речници в .NET.

Реализация на речник с червено-черно дърво

Тъй като имплементацията на речник чрез балансирано дърво е сложна и обширна задача, няма да я разглеждаме във вид на сорс код. Вместо това ще разгледаме класа **`SortedDictionary<K, V>`**, който идва наготово заедно със стандартните библиотеки на .NET. Силно препоръчваме на по-любознателните читатели да разгледат изходния код на SortedDictionary, използвайки някой от инструментите **JustDecompiler** или **ILSpy**, споменати в глава 1.

Както обяснихме в предходната глава, червено-черното дърво е подредено двоично балансирано дърво за претърсване. Ето защо едно от важните изисквания, които са наложени върху множеството от ключове при използването на **`SortedDictionary<K, V>`**, е те **да имат наредба**. Това означава, че ако имаме два ключа, то или единият е по-голям от другия, или те са равни.

Използването на двоично дърво ни носи едно силно предимство: ключовете в речника се пазят сортирани. Благодарение на това свойство, ако данните ни трябват подредени по ключ, няма нужда да ги сортираме допълнително. Всъщност това свойство е единственото предимство на тази реализация пред реализацията с хеш-таблица. Трябва да се подчертае обаче, че пазенето на ключовете сортирани идва със своята цена. Търсенето на елементите с балансирани дървета е по-бавна (сложност O(log n)) от работата с хеш-таблици O(1). По тази причина, ако няма специални изисквания за наредба на ключовете, за предпочитане е да се използва **`Dictionary<K, V>`**. Понятието сложност е обяснено в следващата тема.

Използвайте реализация на речник чрез балансирано дърво само когато се нуждаете от свойството наредените двойки винаги да са сортирани по ключ. Имайте предвид обаче, че балансираното дърво гарантира сложност на търсене, добавяне и изтриване от порядъка на log(n), докато сложността на търсене в хеш-таблицата може да достигне до линейна.

Класът SortedDictionary<K, V>

Класът **`SortedDictionary<K, V>`** представлява имплементация на речник чрез червено-черно дърво. Този клас имплементира всички стандартни операции, дефинирани в интерфейса **`IDictionary<K, V>`**.

Използване на класа SortedDictionary – пример

Сега ще решим един практически проблем, където използването на класа **`SortedDictionary`** е уместно. Нека имаме някакъв текст. Нашата задача ще бъде да намерим всички различни думи в текста, както и колко пъти се среща всяка от тях. Като допълнително условие ще искаме да изведем намерените думи по азбучен ред.

При тази задача използването на речник се оказва особено подходящо. За ключове ще изберем думите от текста, а стойностите срещу всеки ключ в речника, ще бъдат броят срещания на съответната дума.

Алгоритъмът за броене на думите се състои в следното: четем текста дума по дума. За всяка дума проверяваме дали вече присъства в речника. Ако отговорът е не, добавяме нов елемент в речника с ключ думата и стойност 1 (броим първото срещане). Ако отговорът е да - увеличаваме старата стойност с единица за да преброим новото срещане на думата.

Използването на речник реализиран чрез балансирано дърво, ни дава свойството, когато обхождаме елементите му те да бъдат сортирани по ключ. По този начин реализираме допълнително наложеното условие думите да са сортирани по азбучен ред. Следва реализация на описания алгоритъм:

`TreeMapExample.cs`

```
using System;
using System.Collections.Generic;

class TreeMapDemo
{
  private static readonly string TEXT =
    "She uchish li she bachkash li? Be kvo she bachkash " +
    "be? Tui vashto uchene li e? Ia po-hubavo opitai da " +
    "BACHKASH da se uchish malko! Uchish ne uchish trqbva " +
    "da bachkash!";

  static void Main()
  {
    IDictionary<String, int> wordOccurrenceMap =
      GetWordOccurrenceMap(TEXT);
    PrintWordOccurrenceCount(wordOccurrenceMap);
  }

  private static IDictionary<string, int> GetWordOccurrenceMap(
      string text)
   {

      string[] tokens =
         text.Split(' ', '.', ',', '-', '?', '!');

      IDictionary<string, int> words =
         new SortedDictionary<string, int>();

      foreach (string word in tokens)
      {
         if (string.IsNullOrEmpty(word.Trim()))
         {
            continue;
         }

         int count;
         if (!words.TryGetValue(word, out count))
         {
            count = 0;
         }
         words[word] = count + 1;
      }
      return words;
   }

   private static void PrintWordOccurrenceCount(
      IDictionary<string, int> wordOccuranceMap)
   {
      foreach (KeyValuePair<string, int> wordEntry
         in wordOccuranceMap)
      {
         Console.WriteLine(
            "Word '{0}' occurs {1} time(s) in the text",
            wordEntry.Key, wordEntry.Value);
      }

      Console.ReadKey();
   }
}
```

Изходът от примерната програма е следният:

```
Word 'bachkash' occurs 3 time(s) in the text
Word 'BACHKASH' occurs 1 time(s) in the text
Word 'be' occurs 1 time(s) in the text
Word 'Be' occurs 1 time(s) in the text
…
Word 'Tui' occurs 1 time(s) in the text
Word 'uchene' occurs 1 time(s) in the text
Word 'uchish' occurs 3 time(s) in the text
Word 'Uchish' occurs 1 time(s) in the text
Word 'vashto' occurs 1 time(s) in the text
```

В този пример за пръв път демонстрираме обхождане на всички елементи на речник – методът **`PrintWordOccurrenceCount(IDictionary<string, int>)`**. За целта използваме конструкцията за цикъл **`foreach`**. При обхождане на речници, трябва да обърнем внимание, че за разлика от списъците и масивите, елементите на тази структура от данни са наредени двойки (ключ и стойност), а не просто "единични" обекти. Както вече знаем обхождането на елементите на списък с **`foreach`** се свежда до извикване на методи на **`IEnumerable`**, който задължително се имплементира от класа на енумерирания обект. Тъй като **`IDictionary<K, V>`** имплементира интерфейса **`IEnumerable<KeyValuePair<K, V>>`**, това означава, че **`foreach`** итерира върху списък с обекти от тип **`KeyValuePair<K, V>`**.

Интерфейсът IComparable<K>

При използване на **`SortedDictionary<K, V>`** има задължително изискване ключовете да са от тип, чиито стойности могат да се сравняват по големина. В нашия пример ползваме за ключ обекти от тип **`string`**.

Класът **`string`** имплементира интерфейса **`IComparable`**, като сравнението е стандартно (лексикографски). Какво означава това? Тъй като по подразбиране низовете в .NET са case sensitive (т.е. има разлика между главна и малка буква), то думи като "Count" и "count" се смятат за различни, а думите, които започват с малка буква, са преди тези с голяма. Това е следствие от естествената наредба на низовете дефинирана в класа **`string`**. Тази дефиниция идва от имплементацията на метода **`CompareTo(object)`**, чрез който класът **`string`** имплементира интерфейса **`IComparable`**.

Интерфейсът IComparer<T>

Какво можем да направим, когато естествената наредба не ни удовлетворява? Например, ако искаме при сравнението на думите да не се прави разлика между малки и главни букви.

Един вариант е след като прочетем дадена дума да я преобразуваме към малки или главни букви. Този подход ще работи за символни низове, но понякога ситуацията е по-сложна. Затова сега ще покажем друго решение, което работи за всеки произволен клас, който няма естествена наредба (не имплементира **`IComparable`**) или има естествена наредба, но ние искаме да я променим.

За сравнение на обекти по изрично дефинирана наредба в **`SortedDictionary<K, V>`** в .NET се използва интерфейс **`IComparer<T>`**. Той дефинира функция за сравнение **`int Compare(T x, T y)`**, която задава алтернативна на естествената наредба. Нека разгледаме в детайли този интерфейс.

Когато създаваме обект от класа **SortedDictionary<K, V>** можем да подадем на конструктора му референция към **IComparer<K>** и той да използва него при сравнение на ключовете (които са елементи от тип **K**).

Ето една реализация на интерфейса **IComparer<K>**, която променя поведението при сравнение на низове, така че да **не** се различават по големи и малки букви

```
class CaseInsensitiveComparer : IComparer<string>
{
  public int Compare(string s1, string s2)
  {
    return string.Compare(s1, s2, true);
  }
}
```

Нека използваме този **IComparer<E>** при създаването на речника:

```
IDictionary<string, int> words =
  new SortedDictionary<string, int>(
    new CaseInsensitiveComparer());
```

След тази промяна резултатът от изпълнението на програмата ще бъде:

```
Word 'bachkash' occurs 4 time(s) in the text
Word 'Be' occurs 2 time(s) in the text
Word 'da' occurs 3 time(s) in the text
...
Word 'Tui' occurs 1 time(s) in the text
Word 'uchene' occurs 1 time(s) in the text
Word 'uchish' occurs 4 time(s) in the text
Word 'vashto' occurs 1 time(s) in the text
```

Виждаме, че за ключ остава вариантът на думата, който е срещнат за първи път в текста. Това е така, тъй като при извикване на метода **words[word] = count + 1** се подменя само стойността, но не и ключът.

Използвайки **IComparer<E>** ние на практика сменихме дефиницията за подредба на ключове в рамките на нашия речник. Ако за ключ използвахме клас, дефиниран от нас, например **Student**, който имплементира **IComparable<E>**, бихме могли да постигнем същия ефект чрез подмяна на реализацията на метода му **CompareTo(Student)**. Има обаче едно изискване, което трябва винаги да се стремим да спазваме, когато имплементираме **IComparable<K>**. То гласи следното:

Винаги, когато два обекта са еднакви (`Equals(object)` връща `true`), `CompareTo(E)` трябва да връща 0.

Удовлетворяването на това условие ще ни позволи да ползваме обектите от даден клас за ключове, както в реализация с балансирано дърво (**`SortedDictionary`**, конструиран без **`Comparer`**), така и в реализация с хеш-таблица (**`Dictionary`**).

Хеш-таблици

Нека сега се запознаем със структурата от данни хеш-таблица, която реализира по един изключително ефективен начин абстрактната структура данни речник. Ще обясним в детайли как работят хеш-таблиците и защо са толкова ефективни.

Реализация на речник с хеш-таблица

Реализацията с хеш-таблица има важното предимство, че времето за достъп до стойност от речника, при правилно използване, теоретично не зависи от броя на елементите в него.

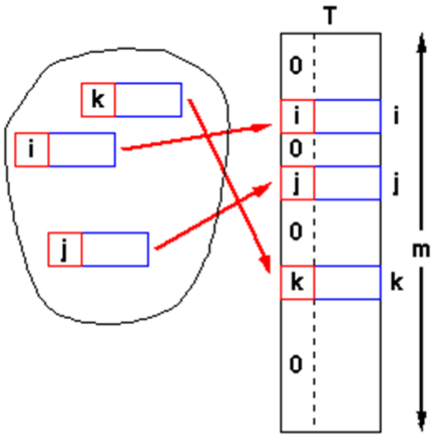
За сравнение да вземем списък с елементи, които са подредени в случаен ред. Искаме да проверим дали даден елемент се намира в него. В най-лошия случай, трябва да проверим всеки един елемент от него, за да дадем категоричен отговор на въпроса "съдържа ли списъкът елемента или не". Очевидно е, че броят на тези сравнения зависи (линейно) от броят на елементите в списъка.

При хеш-таблиците, ако разполагаме с ключ, броят сравнения, които трябва да извършим, за да установим има ли стойност с такъв ключ, е константен и не зависи от броя на елементите в нея. Как точно се постига такава ефективност ще разгледаме в детайли по-долу.

Когато реализациите на някои структури от данни ни дават време за достъп до елементите ѝ, независещ от броя на елементите в нея, се казва, че те притежават свойството **random access** (**свободен достъп**). Такова свойство обикновено се наблюдава при реализации на абстрактни структури от данни с хеш-таблици и масиви.

Какво е хеш-таблица?

Структурата от данни хеш-таблица обикновено се реализира с масив. Тя съдържа наредени двойки (ключ, стойност), които са разположени в масива на пръв поглед случайно и непоследователно. В позициите, в които нямаме наредена двойка, имаме празен елемент (**`null`**):

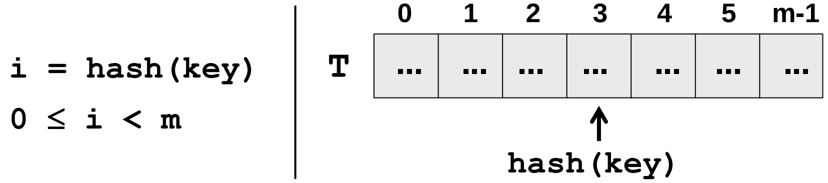


Размерът на таблицата (масива), наричаме **капацитет (capacity)** на хеш-таблицата. **Степен на запълненост (load factor)**, наричаме реално число между 0 и 1, което съответства на отношението между броя на запълнените елементи и текущия капацитет. На фигурата имаме хеш-таблица с 3 елемента и капацитет `m`. Следователно степента на запълване на тази хеш-таблица е `3/m`.

Добавянето и търсенето на елементи става, като върху ключа се приложи някаква функция `hash(key)`, която връща число, наречено **хеш-код**. Като вземем остатъка при деление на този хеш-код с капацитета `m` получаваме число между `0` и `m-1`:

```
index = hash(key) % m
```

На фигурата е показана хеш-таблица `T` с капацитет `m` и хеш-функция `hash(key)`:



Това число ни дава позицията, на която да търсим или добавяме наредената двойка. Ако хеш-функцията разпределя ключовете равномерно, в болшинството случаи на различен ключ ще съответства различна хеш-стойност и по този начин във всяка клетка от масива ще има най-много един ключ. В крайна сметка получаваме изключително бързо търсене и бързо добавяне. Разбира се, може да се случи различни ключове да имат един и същ хеш-код. Това е специален случай, който ще разгледаме след малко.

Използвайте реализация на речник чрез хеш-таблици, когато се нуждаете от максимално бързо намиране на стойностите по ключ.

Капацитетът на таблицата се увеличава, когато броят на наредените двойки в хеш-таблицата стане равен или по-голям от дадена константа, наречена **максимална степен на запълване**. При разширяване на капацитета (най-често удвояване) всички елементи се преподреждат според своя хеш-код и стойността на новия капацитет. Степента на запълване след преподреждане значително намалява. Операцията е времеотнемаща, но се извършва достатъчно рядко, за да не влияе на цялостната производителност на операцията добавяне.

Класът Dictionary<K, V>

Класът `Dictionary<K, V>` е стандартна имплементация на речник с хеш-таблица в .NET Framework. В следващите точки ще разгледаме основните операции, които той предоставя. Ще разгледаме и един конкретен пример, илюстриращ използването на класа и неговите методи.

Основни операции с класа Dictionary<K, V>

Създаването на хеш-таблица става чрез извикването на някои от конструкторите на `Dictionary<K, V>`. Чрез тях можем да зададем начални стойности за капацитет и максимална степен на запълване. Добре е, ако предварително знаем приблизителният брой на елементите, които ще бъдат добавени в нашата хеш-таблица, да го укажем още при създаването й. Така ще избегнем излишното разширяване на таблицата и ще постигнем по-добра ефективност. По подразбиране стойността на началния капацитет е 16, а на максималната степен на запълване е 0.75.

Да разгледаме какво прави всеки един от методите реализирани в класа `Dictionary<K, V>`:

- `void Add(K, V)` добавя нова стойност за даден ключ. При опит за добавяне на ключ, който вече съществува в речника, се хвърля изключение. Операцията работи изключително бързо.
- `bool TryGetValue(K, out V)` връща елемент от тип V чрез `out` параметър за дадения ключ или `null`, ако няма елемент с такъв ключ. Резултатът от изпълнението на метода е `true` ако е намерен елемент. Операцията е много бърза, тъй като алгоритъмът за търсене на елемент по ключ в хеш-таблица се доближава по сложност до O(1)
- `bool Remove(K)` изтрива от речника елемента с този ключ. Операцията работи изключително бързо.
- `void Clear()` премахва всички елементи от речника.

- **bool ContainsKey(K)** проверява дали в речника присъства наредена двойка с посочения ключ. Операцията работи изключително бързо.
- **bool ContainsValue(V)** проверява дали в речника присъстват една или повече наредени двойки с посочената стойност. Тази операция работи бавно, тъй като проверява всеки елемент на хеш-таблицата.
- **int Count** връща броя на наредените двойки в речника.
- Други операции – например извличане на всички ключове, стойности или наредени двойки в структура, която може да бъде обходена чрез цикъл.

Студенти и оценки – пример

Ще илюстрираме как се използват някои от описаните по-горе операции чрез един пример. Нека имаме студенти, като всеки от тях би могъл да има най-много една оценка. Искаме да съхраняваме оценките в някаква структура, в която можем бързо да търсим по име на студент.

За тази задача ще създадем хеш-таблица с начален капацитет 6. Тя ще има за ключове имената на студентите, а за стойности – някакви техни оценки. Добавяме 6 примерни студента, след което наблюдаваме какво се случва когато отпечатваме на стандартния изход техните данни. Ето как изглежда кодът от този пример:

```
using System;
using System.Collections.Generic;

class StudentsExample
{
  static void Main()
  {
    IDictionary<string, double> studentMarks =
    new Dictionary<string, double>();

    studentMarks["Pesho"] = 3.00;
    studentMarks["Gosho"] = 4.50;
    studentMarks["Nakov"] = 5.50;
    studentMarks["Vesko"] = 3.50;
    studentMarks["Tsanev"] = 4.00;
    studentMarks["Nerdy"] = 6.00;

    double tsanevMark = studentMarks["Tsanev"];
    Console.WriteLine("Tsanev's mark: {0:0.00}", tsanevMark);

    studentMarks.Remove("Tsanev");
    Console.WriteLine("Tsanev's mark removed.");

    Console.WriteLine("Is Tsanev in the dictionary: {0}",
    studentMarks.ContainsKey("Tsanev") ? "Yes!": "No!");
```

```
        Console.WriteLine("Nerdy's mark is {0:0.00}.",
            studentMarks["Nerdy"]);
        studentMarks["Nerdy"] = 3.25;
        Console.WriteLine(
            "But we all know he deserves no more than {0:0.00}.",
            studentMarks["Nerdy"]);

        double mishosMark;
        bool findMisho = studentMarks.TryGetValue("Misho",
            out mishosMark);

        Console.WriteLine(
            "Is Misho's mark in the dictionary? {0}",
            findMisho ? "Yes!": "No!");

        studentMarks["Misho"] = 6.00;
        findMisho = studentMarks.TryGetValue("Misho",
            out mishosMark);

        Console.WriteLine(
            "Let's try again: {0}. Misho's mark is {1}",
            findMisho ? "Yes!" : "No!", mishosMark);

        Console.WriteLine("Students and marks:");

        foreach (KeyValuePair<string, double> studentMark
            in studentMarks)
        {
            Console.WriteLine("{0} has {1:0.00}",
                studentMark.Key, studentMark.Value);
        }

        Console.WriteLine(
            "There are {0} students in the dictionary",
            studentMarks.Count);
        studentMarks.Clear();
        Console.WriteLine("Students dictionary cleared.");
        Console.WriteLine("Is dictionary empty: {0}",
            studentMarks.Count == 0);

        Console.ReadLine();
    }
}
```

Изходът от изпълнението на този код е следният:

```
Tsanev's mark: 4.00
Tsanev's mark removed.
Is Tsanev in the dictionary: No!
Nerdy's mark is 6.00.
```

```
But we all know he deserves no more than 3.25.
Is Misho's mark in the dictionary? No!
Let's try again: Yes!. Misho's mark is 6
Students and marks:
Pesho has 3.00
Gosho has 4.50
Nakov has 5.50
Vesko has 3.50
Misho has 6.00
Nerdy has 3.25
There are 6 students in the dictionary
Students dictionary cleared.
Is dictionary empty: True
```

Виждаме, че няма подредба при отпечатването на студентите. Това е така, защото при хеш-таблиците (за разлика от балансираните дървета) елементите **не се** пазят сортирани. Дори ако текущият капацитет на таблицата се промени докато работим с нея, много е вероятно да се промени и редът, в който се пазят наредените двойки. Причината за това поведение ще анализираме по-долу.

Важно е да се запомни, че при хеш-таблиците не можем да разчитаме на никаква наредба на елементите. Ако се нуждаем от такава, можем преди отпечатване да сортираме елементите. Друг вариант е да използваме **`SortedDictionary<K, V>`**.

Хеш-функции и хеширане

Сега ще се спрем по-детайлно на понятието, хеш-код, което употребихме малко по-рано. Хеш-кодът представлява числото, което ни връща т.нар. **хеш-функция**, приложена върху ключа. Това число трябва да е различно за всеки различен ключ или поне с голяма вероятност при различни ключове хеш-кодът трябва да е различен.

Хеш-функции

Съществува понятието **перфектна хеш-функция** (perfect hash function). Една хеш-функция се нарича перфектна, ако при N ключа, на всеки ключ функцията съпоставя различно цяло число в някакъв смислен интервал (например от 0 до N-1). Намирането на такава функция в общия случай е доста трудна, почти невъзможна задача. Такива функции си струва да се използват само при множества от ключове, които са с предварително известни елементи или поне ако множеството от ключове рядко се променя.

В практиката се използват други, не чак толкова "перфектни" хеш-функции.

Сега ще разгледаме няколко примера за хеш-функции, които се използват директно в .NET библиотеките.

Методът GetHashCode() в .NET платформата

Всички .NET класове имат метод **`GetHashCode()`**, който връща стойност от тип **`int`**. Този метод се наследява от класа **`Object`**, който стои в корена на йерархията на всички .NET класове.

Имплементацията в класа **`Object`** на метода **`GetHashCode()`** е такава, че **не се** гарантира уникалността на резултата. Това означава, че класовете наследници трябва да осигурят имплементация на **`GetHashCode()`** за да се ползват за ключ на хеш-таблица.

Друг пример за хеш-функция, която идва директно в .NET, е използваната от класовете **`int`**, **`byte`** и **`short`** (дефиниращи целите числа). Там за хеш-код се ползва стойността на самото число.

Един по-сложен пример за хеш-функция е имплементацията на класа **`string`**:

```
public override unsafe int GetHashCode()
{
   fixed (char* str = ((char*)this))
   {
      char* chPtr = str;
      int num = 352654597;
      int num2 = num;
      int* numPtr = (int*)chPtr;
      for (int i = this.Length; i > 0; i -= 4)
      {
         num = (((num << 5) + num) + (num >> 27)) ^ numPtr[0];
         if (i <= 2)
         {
            break;
         }
         num2 = (((num2 << 5) + num2) + (num2 >> 27)) ^
                numPtr[1];

         numPtr += 2;
      }
      return (num + (num2 * 1566083941));
   }
}
```

Имплементацията е доста сложна, но това, което трябва да запомним е, че тя гарантира уникалността на резултата точно когато низовете са различни. Още нещо което може да забележим е, че сложността на алгоритъма за изчисляване на хеш-кода на **`string`** е пропорционална на **`Length / 4`** или O(n), което означава, че колкото по-дълъг е низа толкова по-бавно ще се изчислява неговия хеш-код.

На читателя оставяме да разгледа други имплементации на метода **`GetHashCode()`** в някои от най-често използваните класове като **`Date`**, **`long`**, **`float`** и **`double`**.

Сега, нека се спрем на въпроса как да имплементираме сами този метод за нашите класове. Вече обяснихме, че оставянето на имплементацията, която идва наготово от **`object`**, не е допустимо решение. Друга много проста имплементация е винаги да връщаме някаква фиксирана константа, примерно:

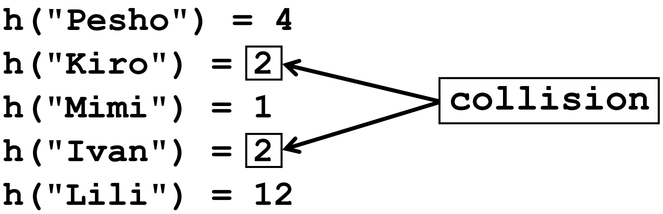
```
public override int GetHashCode()
{
  return 42;
}
```

Ако в хеш-таблица използваме за ключове обекти от клас, който има горната имплементация на **`GetHashCode()`**, ще получим много лоша производителност, защото всеки път, когато добавяме нов елемент в таблицата, ще трябва да го слагаме на едно и също място. Когато търсим, всеки път ще попадаме в една и съща клетка на таблицата.

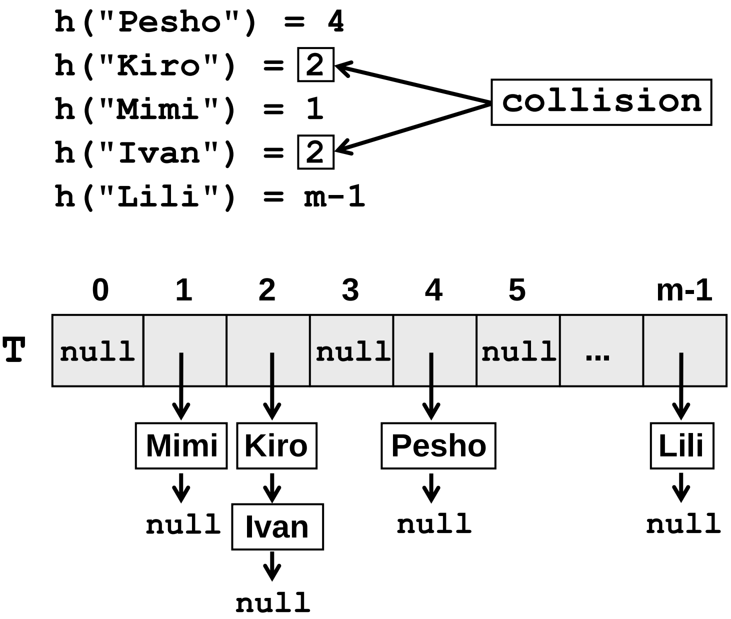
За да се избягва описаното неблагоприятно поведение, трябва хеш-функцията да разпределя ключовете равномерно сред възможните стойности за хеш-код.

Колизии при хеш-функциите

Ситуация, при която два различни ключа връщат едно и също число за хеш-код наричаме **колизия**:



Как да решим проблема с колизиите ще разгледаме подробно в следващия параграф. Най-простото решение, обаче е очевидно: двойките, които имат ключове с еднакви хеш-кодове да нареждаме в списък:



Следователно при използване на константа 42 за хеш-код, нашата хеш-таблица се изражда в линеен списък и употребата ѝ става неефективна.

Имплементиране на метода GetHashCode()

Ще дадем един стандартен алгоритъм, по който можем сами да имплементираме `GetHashCode()`, когато ни се наложи:

Първо трябва да определим полетата на класа, които участват по някакъв начин в имплементацията на `Equals(object)` метода. Това е необходимо, тъй като винаги, когато `Equals()` е `true` трябва резултатът от `GetHashCode()` да е един и същ. Така полетата, които не участват в пресмятането на `Equals()`, не трябва да участват и в изчисляване на `GetHashCode()`.

След като сме определили полетата, които ще участват в изчислението на `GetHashCode()`, трябва по някакъв начин да получим за тях стойности от тип `int`. Ето една примерна схема:

- Ако полето е `bool`, за `true` взимаме `1`, а за `false` взимаме `0` (или директно викаме `GetHashCode()` на `bool`).
- Ако полето е от тип `int`, `byte`, `short`, `char` можем да го преобразуваме към `int`, чрез оператора за явно преобразуване `(int)` (или директно викаме `GetHashCode()`).
- Ако полето е от тип `long`, `float` или `double`, можем да ползваме наготово резултата от техните `GetHashCode()`.

- Ако полето не е от примитивен тип, просто извикваме метода **`GetHashCode()`** на този обект. Ако стойността на полето е **`null`**, връщаме `0`.
- Ако полето е масив или някаква колекция, извличаме хеш-кода за всеки елемент на тази колекция.

Накрая сумираме получените **`int`** стойности, като преди всяко събиране умножаваме временния резултат с някое просто число (например 83), като игнорираме евентуалните препълвания на типа **`int`**.

В крайна сметка получаваме хеш-код, който е добре разпределен в пространството от всички 32-битови стойности. Можем да очакваме, че при така изчислен хеш-код колизиите ще са рядкост, тъй като всяка промяна в някое от полетата, участващи в описаната схема за изчисление, води до съществена промяна в хеш-кода.

Имплементиране на GetHashCode() – пример

Да илюстрираме горният алгоритъм с един пример. Нека имаме клас, чиито обекти представляват точка в тримерното пространство. И нека точката вътрешно представяме чрез нейните координати по трите измерения x, y и z:

Point3D.cs

```
/// <summary>
/// Class representing a point in three dimensional space.
/// </summary>
public class Point3D
{
   private double x;
   private double y;
   private double z;

   /// <summary>
   /// Constructs a new <see cref="Point3D"/> instance
   /// with the specified Cartesian coordinates of the point.
   /// </summary>
   /// <param name="x">x coordinate of the point</param>
   /// <param name="y">y coordinate of the point</param>
   /// <param name="z">z coordinate of the point</param>
   public Point3D(double x, double y, double z)
   {
      this.x = x;
      this.y = y;
      this.z = z;
   }
}
```

Можем лесно да реализираме **`GetHashCode()`** по описания по-горе алгоритъм:

```
...
  public override bool Equals(object obj)
  {
    if (this == obj)
      return true;

    Point3D other = obj as Point3D;

    if (other == null)
      return false;

    if (!this.x.Equals(other.x))
      return false;

    if (!this.y.Equals(other.y))
      return false;

    if (!this.z.Equals(other.z))
      return false;

    return true;
  }


  public override int GetHashCode()
  {
    int prime = 83;
    int result = 1;
    unchecked
    {
      result = result * prime + x.GetHashCode();
      result = result * prime + y.GetHashCode();
      result = result * prime + z.GetHashCode();
    }

    return result;
  }
}
```

Тази имплементация е несравнимо по-добра, от това да не правим нищо или да връщаме константа. Въпреки това колизиите и при нея се срещат, но доста по-рядко.

Интерфейсът IEqualityComparer<T>

Едно от най-важните неща, които разбрахме досега е, че за да ползваме инстанциите на даден клас като ключове за речник, то класа трябва да имплементира правилно **`GetHashCode`** и **`Equals`**. Но какво да направим, ако искаме да използваме клас, който не можем или не искаме да наследим или променим? В този случай на помощ идва интерфейсът **IEqualityComparer<T>**. Той дефинира следните две операции:

bool Equals(T obj1, T obj2) – връща **true** ако **obj1** и **obj2** са равни

int GetHashCode(T obj) – връща хеш-кода за дадения обект.

Вече се досещате, че речниците в .NET могат да използват инстанция на **IEqualityComparer**, вместо съответните методи на класа даден за ключ. По този начин разработчиците могат да ползват практически всеки клас за ключ на речник, стига да осигурят имплементация на **IEqualityComparer** за този клас. Дори нещо повече - когато предоставим **IEqualityComparer** на речник, можем да променим начина, по който се изчислява **GetHashCode** и **Equals** за всякакви типове, дори за тези в .NET, тъй като в този случай речника използва методите на интерфейса вместо съответните методи на класа ключ. Ето един пример за имплементация на **IEqualityComparer**, за класа Point3D, който разгледахме по-рано:

```
public class Point3DEqualityComparer : IEqualityComparer<Point3D>
{

   #region IEqualityComparer Members

   public bool Equals(Point3D point1, Point3D point2)
   {
      if (point1 == point2)
         return true;

      if (point1 == null ^ point2 == null)
         return false;

      if (!point1.X.Equals(point2.X))
         return false;

      if (!point1.Y.Equals(point2.Y))
         return false;

      if (!point1.Z.Equals(point2.Z))
         return false;

      return true;
   }

   public int GetHashCode(Point3D obj)
   {
      Point3D point = obj as Point3D;
      if (point == null)
      {
         return 0;
      }
```

```
        int prime = 83;
        int result = 1;
        unchecked
        {
            result = result * prime + point.X.GetHashCode();
            result = result * prime + point.Y.GetHashCode();
            result = result * prime + point.Z.GetHashCode();
        }
        return result;
    }

    #endregion
}
```

За да използваме **`Point3DEqualityComparer`** е достатъчно единствено да го подадем като параметър на конструктора на нашия речник:

```
IEqualityComparer<Point3D> comparer =
    new Point3DEqualityComparer();

Dictionary<Point3D, int> dict =
    new Dictionary<Point3D, int>(comparer);

dict[new Point3D(1, 2, 3)] = 1;
Console.WriteLine(++dict[new Point3D(1, 2, 3)]);
```

Решаване на проблема с колизиите

На практика колизиите има почти винаги с изключение на много редки и специфични ситуации. За това е необходимо да живеем с идеята за тяхното присъствие в нашите хеш-таблици и да се съобразяваме с тях. Нека разгледаме няколко стратегии за справяне с колизиите:

Нареждане в списък (chaining)

Най-разпространеният начин за решаване на проблема с колизиите е нареждането в списък (chaining). Той се състои в това двойките ключ и стойност, които имат еднакъв хеш-код за ключа да се нареждат в списък един след друг.

Реализация на речник чрез хеш-таблица и chaining

Нека си поставим за задача да реализираме структурата от данни речник чрез хеш-таблица с решаване на колизиите чрез нареждане в списък (chaining). Да видим как може да стане това. Първо ще дефинираме клас, който описва наредената двойка (Key, Value):

KeyValuePair.cs

```
public struct KeyValuePair<TKey, TValue>
{
  private TKey key;
  private TValue value;

  public KeyValuePair(TKey key, TValue value)
  {
    this.key = key;
    this.value = value;
  }

  public TKey Key
  {
    get
    {
      return this.key;
    }
  }

  public TValue Value
  {
    get
    {
      return this.value;
    }
  }

  public override string ToString()
  {
    StringBuilder builder = new StringBuilder();
    builder.Append('[');
    if (this.Key != null)
    {
      builder.Append(this.Key.ToString());
    }
    builder.Append(", ");
    if (this.Value != null)
    {
      builder.Append(this.Value.ToString());
    }
    builder.Append(']');
    return builder.ToString();
  }
}
```

Този клас има конструктор, който приема ключ от тип **`TKey`** и стойност от тип **`TValue`**. Дефинирани са два метода за достъп съответно за ключа (**`Key`**) и стойността (**`Value`**). Ще отбележим, че нарочно нямаме публични методи, чрез които да променяме стойностите на ключа и стойността. Това прави този клас непроменяем (immutable). Това е добра идея, тъй като обектите, които ще се пазят вътрешно в реализациите на речника, ще бъдат същите като тези, които ще връщаме например при реализацията на метод за вземане на всички наредени двойки.

Предефинирали сме метода **`ToString()`**, за да можем лесно да отпечатваме наредената двойка на стандартния изход или в текстов файл.

Следва примерен шаблонен интерфейс, който дефинира най-типичните операции за типа речник:

Idictionary.cs

```
using System;
using System.Collections.Generic;

/// <summary>
/// Interface that defines basic methods needed
/// for a class which maps keys to values.
/// </summary>
/// <typeparam name="K">Key type</typeparam>
/// <typeparam name="V">Value type</typeparam>
public interface IDictionary<K, V> :
  IEnumerable<KeyValuePair<K, V>>
{

  /// <summary>
  /// Adds the specified value by the specified
  /// key to the dictionary. If the key already
  /// exists its value is replaced with the
  /// new value and the old value is returned.
  /// </summary>
  /// <param name="key">Key for the new value</param>
  /// <param name="value">Value to be mapped
  /// with that key</param>
  /// <returns>the old value for the specified
  /// key or null if the key does not exist</returns>
  /// <exception cref="NullReferenceException">
  /// If Key is null</exception>
  V Set(K key, V value);

  ///<summary>
  /// Finds the value mapped by specified key.
  ///</summary>
  /// <param name="key">key for which the value
  /// is needed.</param>
  /// <returns>value for the specified key if present,
  /// or null if there is no value with such key</returns>
  V Get(K key);
```

```
    /// <summary>
    /// Gets or sets the value of the entry in the
    /// dictionary identified by the Key specified.
    /// </summary>
    /// <remarks>A new entry will be created if the
    /// value is set for a key that is not currently
    /// in the Dictionary</remarks>
    /// <param name="key">Key to identify the entry
    /// in the Dictionary</param>
    /// <returns>Value of the entry in the Dictionary
    /// identified by the Key provided</returns>
    V this[K key] { get; set; }

    /// <summary>
    /// Removes an element in the Dictionary
    /// identified by a specified key.
    /// </summary>
    /// <param name="key">Key to identify the
    /// element in the Dictionary to be removed</param>
    /// <returns></returns>
    bool Remove(K key);

    /// <summary>
    /// Get a value indicating the number of
    /// entries in the Dictionary
    /// </summary>
    /// <returns></returns>
    int Count { get; }

    /// <summary>
    /// Removes all the elements from the dictionary.
    /// </summary>
    void Clear();
}
```

В интерфейса по-горе, както и в предходния клас използваме шаблонни типове (generics), чрез които декларираме параметри за типа на ключовете (**K**) и типа на стойностите (**V**). Това позволява нашият речник да бъде използван с произволни типове за ключовете и за стойностите. Както вече знаем, единственото изискване е ключовете да дефинират коректно методите **Equals()** и **GetHashCode()**.

Нашият интерфейс **IDictionary<K, V>** прилича много на интерфейса **System.Collections.Generic.IDictionary<K, V>**, но е по-прост от него и описва само най-важните операции върху типа данни "речник". Той наследява системния интерфейс **IEnumerable<DictionaryEntry<K, V>>**, за да позволи речникът да бъде обхождан във for цикъл.

Следва примерна имплементация на речник, при който проблемът с колизиите се решава чрез нареждане в списък (chaining):

HashDictionary.cs

```
/// <summary>
/// Implementation of <see cref="IDictionary"/> interface
/// using hash table. Collisions are resolved by chaining.
/// </summary>
/// <typeparam name="K">Type of the keys</typeparam>
/// <typeparam name="V">Type of the values</typeparam>
public class HashDictionary<K, V>:IEnumerable<KeyValuePair<K, V>>
{
  private const int DEFAULT_CAPACITY = 2;
  private const float DEFAULT_LOAD_FACTOR = 0.75f;
  private List<KeyValuePair<K, V>>[] table;
  private float loadFactor;
  private int threshold;
  private int size;
  private int initialCapacity;

  public HashDictionary()
    :this(DEFAULT_CAPACITY, DEFAULT_LOAD_FACTOR)
  {
  }

  private HashDictionary(int capacity, float loadFactor)
  {
    this.initialCapacity = capacity;
    this.table = new List<KeyValuePair<K, V>>[capacity];
    this.loadFactor = loadFactor;
    unchecked
    {
      this.threshold =
      (int)(capacity * this.loadFactor);
    }
  }

  public void Clear()
  {
    if (this.table != null)
    {
      this.table =
        new List<KeyValuePair<K, V>>[initialCapacity];
    }
    this.size = 0;
  }

  private List<KeyValuePair<K, V>> FindChain(
    K key, bool createIfMissing)
  {
    int index = key.GetHashCode();
    index = index % this.table.Length;
```

```
        if (this.table[index] == null && createIfMissing)
        {
            this.table[index] = new List<KeyValuePair<K, V>>();
        }
        return this.table[index] as List<KeyValuePair<K, V>>;
    }

    public V Get(K key)
    {
        List<KeyValuePair<K, V>> chain =
            this.FindChain(key, false);
        if (chain != null)
        {
            foreach (KeyValuePair<K, V> entry in chain)
            {
                if (entry.Key.Equals(key))
                {
                    return entry.Value;
                }
            }
        }

        return default(V);
    }

    public V this[K key]
    {
        get
        {
            return this.Get(key);
        }
        set
        {
            this.Set(key, value);
        }
    }

    public int Count
    {
        get
        {
            return this.size;
        }
    }

    public V Set(K key, V value)
    {
        List<KeyValuePair<K, V>> chain =
            this.FindChain(key, true);
```

```
  for (int i = 0; i < chain.Count; i++)
  {
    KeyValuePair<K, V> entry = chain[i];
    if (entry.Key.Equals(key))
    {
      // Key found -> replace its value
      //with the new value
      KeyValuePair<K, V> newEntry =
        new KeyValuePair<K, V>(key, value);
      chain[i] = newEntry;
      return entry.Value;
    }
  }
  chain.Add(new KeyValuePair<K, V>(key, value));
  if (size++ >= threshold)
  {
    this.Expand();
  }

  return default(V);
}

/// <summary>
/// Expands the underling table
/// </summary>
private void Expand()
{
  int newCapacity = 2 * this.table.Length;
  List<KeyValuePair<K, V>>[] oldTable = this.table;
  this.table = new List<KeyValuePair<K, V>>[newCapacity];
  this.threshold = (int)(newCapacity * this.loadFactor);
  foreach (List<KeyValuePair<K, V>> oldChain in oldTable)
  {
    if (oldChain != null)
    {
      foreach (KeyValuePair<K, V> keyValuePair
        in oldChain)
      {
        List<KeyValuePair<K, V>> chain =
          FindChain(keyValuePair.Key, true);
        chain.Add(keyValuePair);
      }
    }
  }
}

public bool Remove(K key)
{
  List<KeyValuePair<K, V>> chain =
    this.FindChain(key, false);
```

```
        if (chain != null)
        {
            for (int i = 0; i < chain.Count; i++)
            {
                KeyValuePair<K, V> entry = chain[i];
                if (entry.Key.Equals(key))
                {
                    // Key found -> remove it
                    chain.RemoveAt(i);
                    return true;
                }
            }
        }
        return false;
    }

    IEnumerator<KeyValuePair<K, V>>
        IEnumerable<KeyValuePair<K, V>>.GetEnumerator()
    {
        foreach (List<KeyValuePair<K, V>> chain in this.table)
        {
            if (chain != null)
            {
                foreach (KeyValuePair<K, V> entry in chain)
                {
                    yield return entry;
                }
            }
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return ((IEnumerable<KeyValuePair<K, V>>)this).
            GetEnumerator();
    }
}
```

Ще обърнем внимание на по-важните моменти в този код. Нека започнем от конструктора. Единственият публичен конструктор е конструкторът по подразбиране. Той в себе си извиква друг конструктор като му подава някакви предварително зададени стойности за капацитет и степен на запълване. На читателя предоставяме да реализира валидация на тези параметри и да направи и този конструктор публичен, за да предостави повече гъвкавост на ползвателите на този клас.

Следващото нещо, на което ще обърнем внимание, е това как е реализирано нареждането в списък. При конструирането на хеш-таблицата в конструктора инициализираме масив от списъци, които ще съдържат нашите

KeyValuePair обекти. За вътрешно ползване сме реализирали един метод **FindChain()**, който изчислява хеш-кода на ключа като вика метода **GetHashCode()** и след това разделя върнатата хеш-стойност на дължината на таблицата (капацитета). Така се получава индексът на текущия ключ в масива, съхраняващ елементите на хеш-таблицата. Списъкът с всички елементи, имащи съответния хеш-код, се намира в масива на изчисления индекс. Ако списъкът е празен, той има стойност **null**. В противен случай в съответната позиция има списък от елементи за съответния ключ.

На метода **FindChain()** се подава специален параметър, който указва дали да създава празен списък, ако за подадения ключ все още няма списък с елементи. Това предоставя удобство на методите за добавяне на елементи и за преоразмеряване на хеш-таблицата.

Другото нещо, на което ще обърнем внимание, е методът **Expand()**, който разширява текущата таблица, когато се достигне максималното допустимо запълване. За целта създаваме нова таблица (масив), двойно по-голяма от старата. Изчисляваме новото максимално допустимо запълване, това е полето **threshold**. Следва най-важната част. Разширили сме таблицата и по този начин сме сменили стойността на **this.table.Length**. Ако потърсим някой елемент, който вече сме добавили, методът **FindChain(K key)**, изобщо няма да върне правилната верига, в която да го търсим. Затова се налага всички елементи от старата таблица да се прехвърлят, като не просто се копират веригите, а се добавят наново обектите от клас **KeyValuePair** в новосъздадени вериги.

За да имплементираме коректно обхождането на хеш-таблицата, реализирахме интерфейса **IEnumerable<KeyValuePair<K, V>>**, който има метод **GetEnumerator()**, връщащ итератор (**IEnumerator**) по елементите на хеш-таблицата, който в случая за улеснение реализирахме чрез израза **yield return**. След малко ще дадем пример как можем да използваме нашата реализация на хеш-таблица и нейният итератор. Но първо, за да тестваме по-лесно всичките им аспекти, нека направим малка промяна в имплементацията на Point3D, която разгледахме по-рано и по-точно в това как се изчислява хеш-кода:

```
/// <summary>
/// Class representing a point in three dimensional space.
/// </summary>
public class Point3D
{
  private double x;
  private double y;
  private double z;

  /// <summary>
  /// Constructs a new <see cref="Point3D"/> instance
  /// with the specified Cartesian coordinates of the point.
  /// </summary>
/// <param name="x">x coordinate of the point</param>
/// <param name="y">y coordinate of the point</param>
/// <param name="z">z coordinate of the point</param>
public Point3D(double x, double y, double z)
{
  this.x = x;
  this.y = y;
  this.z = z;
}

public double X
{
  get { return x; }
  set { x = value; }
}

public double Y
{
  get { return y; }
  set { y = value; }
}

public double Z
{
  get { return z; }
  set { z = value; }
}

public override bool Equals(object obj)
{
  if (this == obj)
    return true;

  Point3D other = obj as Point3D;

  if (other == null)
    return false;

  if (!this.x.Equals(other.x))
    return false;

  if (!this.y.Equals(other.y))
    return false;

  if (!this.z.Equals(other.z))
    return false;

  return true;
}
```

```

   public override int GetHashCode()
   {
      //int prime = 83;
      //int result = 1;
      //unchecked
      //{
      // result = result * prime + x.GetHashCode();
      // result = result * prime + y.GetHashCode();
      // result = result * prime + z.GetHashCode();
      //}

      int result = (int)Math.Round((x + y + z));

      return result;
   }

   public override string ToString()
   {
      return string.Format("X={0} Y={1} Z={2}", x, y, z);
   }
}
```

В този случай използваме много примитивен подход за генериране на хеш-код, който в случая ще ни помогне да тестваме колизиите в нашата хеш-таблица. Следва примерът, който ползва **HashDictionary** и модифицираната версия на **Point3D**:

```
class Program
{
   static void Main()
   {
      HashDictionary<Point3D, int> dict =
         new HashDictionary<Point3D, int>();

      dict[new Point3D(1, 2, 3)] = 1; //Set value
      Console.WriteLine(dict[new Point3D(1, 2, 3)]); //Get value

      //Set Value, overwrite previous value for the same Key
      dict[new Point3D(1, 2, 3)] += 1;
      Console.WriteLine(dict[new Point3D(1, 2, 3)]);

      //Now this Point has the same HashCode as the previos one
      dict[new Point3D(3, 2, 1)] = 42;

      Console.WriteLine(dict[new Point3D(3, 2, 1)]);
      //test if chaining works and elements with equal
      //hashcodes are not overwritten
      Console.WriteLine(dict[new Point3D(1, 2, 3)]);

      //HashCode to test the creation of another entry
      //in the internal table
      dict[new Point3D(1001, 100, 10)] = 1111;
      Console.WriteLine(dict[new Point3D(1001, 100, 10)]);

      //iterate through the Dictionary entries and print values
      foreach (KeyValuePair<Point3D, int> entry in dict)
      {
        Console.WriteLine(
          "Key: " + entry.Key + "; Value: " + entry.Value);
      }
    }
}
```

Както очаквате, резултатът от изпълнението на програмата е следният:

```
1
2
42
2
1111
Key: X=1 Y=2 Z=3; Value: 2
Key: X=3 Y=2 Z=1; Value: 42
Key: X=1001 Y=100 Z=10; Value: 1111
```

В примерната имплементация на хеш-таблица има още една особеност. Методът **FindChain()** не е реализиран напълно коректно. В повечето случаи тази реализация ще работи без проблем, но какво ще стане, ако добавяме елементи до безкрай? В един прекрасен момент, когато капацитетът е станал 2^{31} и се наложи да го разширим, то при умножение на това число с 2 ще получим -2 (вж. секцията за представяне на отрицателни числа в главата "Бройни системи"). След това при опит за създаване на нов масив с размер -2 естествено ще бъде хвърлено изключение и изпълнението на метода ще бъде прекратено. Нека не лишаваме читателя от удоволствието да прецени как да се справи с тази задача.

Методи за решаване на колизиите от тип отворена адресация (open addressing)

Нека сега разгледаме методите за разрешаване на колизиите, алтернативни на нареждането в списък. Най-общо идеята при тях е, че в случай на колизия се опитваме да сложим новата двойка на някоя свободна позиция от таблицата. Методите се различават по това как се избира къде да се търси свободно място за новата двойка. Освен това трябва да е възможно и намирането на тази двойка на новото ѝ място.

Основен недостатък на този тип методи спрямо нареждането в списък е, че са неефективни при голяма степен на запълненост (близка до 1).

Линейно пробване (linear probing)

Този метод е един от най-лесните за имплементация. Линейното пробване най-общо представлява следният простичък код:

```
int newPosition = (oldPosition + i) % capacity;
```

Тук **`capacity`** е капацитетът на таблицата, **`oldPostion`** е позицията, за която получаваме колизия, а **`i`** е номер на поредното пробване. Ако новополучената позиция е свободна, то мястото се използва за новодобавената двойка, в противен случай пробваме отново, като увеличаваме **`i`** с единица. Възможно е пробването да е както напред така и назад. Пробване назад става като вместо да прибавяме, вадим **`i`** от позицията, в която имаме колизия.

Предимство на този метод е сравнително бързото намиране на нова позиция. За нещастие има изключително висока вероятност, ако на едно място е имало колизия, след време да има и още. Това на практика води до силна неефективност.

Използването на линейно пробване като метод за решаване на проблема с колизиите е неефективно и трябва да се избягва.

Квадратично пробване (Quadratic probing)

Това е класически метод за решаване на проблема с колизиите. Той се различава от линейното пробване с това, че за намирането на нова позиция се използва квадратна функция на **`i`** (номер на поредно пробване). Ето как би изглеждало едно такова решение:

```
int newPosition = (oldPosition + c1*i + c2*i*i) % capacity;
```

Тук се появяват две константи **`c1`** и **`c2`**. Иска се **`c2`** да е различна от 0, защото в противен случай се връщаме на линейно пробване.

От избора на **`c1`** и **`c2`** зависи на кои позиции спрямо началната ще пробваме. Например, ако **`c1`** и **`c2`** са равни на 1, ще пробваме последователно **`oldPosition`**, **`oldPosition + 2`**, **`oldPosition + 6`**, …. За таблица с капацитет от вида **`2n`**, е най-добре да се изберат **`c1`** и **`c2`** равни на **`0.5`**.

Квадратичното пробване е по-ефективно от линейното.

Двойно хеширане (double hashing)

Както става ясно и от името на този метод, при повторното хеширане за намиране на нова позиция се прави повторно хеширане на получения хеш-код, но с друга хеш-функция, съвсем различна от първата. Този метод е по-добър от линейното и квадратичното пробване, тъй като всяко следващо пробване зависи от стойността на ключа, а не от позицията определена за ключа в таблицата. Това има смисъл, защото позицията за даден ключ зависи от текущия капацитет на таблицата.

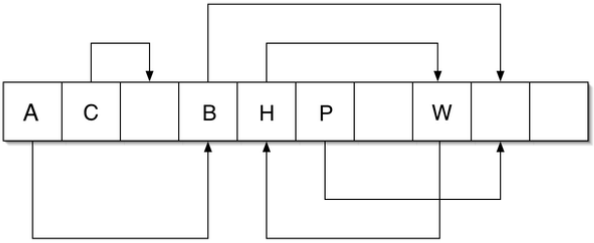
Кукувиче хеширане (cuckoo hashing)

Кукувичето хеширане е сравнително нов метод с отворена адресация за справяне с колизиите. Той е бил представен за пръв път от R. Pagh и F. Rodler през 2001 година. Името му идва от поведението, наблюдавано при някои видове кукувици. Майките кукувици избутват яйца и/или малките на други птици извън гнездото им, за да оставят техните яйца там и така други птици да се грижат за техните яйца (и малки след излюпването).

Основната идея на този метод е да се използват две хеш-функции вместо една. По този начин ще разполагаме не с една, а с две позиции, на които можем да поставим елемент в речника. Ако единият от двата елемента е свободен, то просто слагаме елемента на свободна позиция. Ако пък и двете позиции са заети, то слагаме новият елемент на една от двете позиции, като той "изритва" елемента, който до сега се е намирал там. На свой ред "изритания" елемент отива на своята алтернативна позиция, като "изритва" някой друг елемент, ако е необходимо. Новият "изритан" повтаря процедурата и така, докато не се достигне свободна позиция или докато не се получи зацикляне. Във втория случай цялата таблица се построява наново с по-голям размер и с нови хеш-функции.

На картинката по-долу е показана примерна схема на хеш-таблица, която използва кукувиче хеширане. Всяка клетка, която съдържа елемент има връзка към алтернативната клетка за ключа, който се намира в нея. Сега ще проиграем различни ситуации за добавяне на нов елемент.

Ако поне една от двете хеш-функции ни даде свободна клетка, то няма проблем. Слагаме елемента в една от двете. Нека обаче и двете хеш-функции са дали заети клетки и на случаен принцип сме избрали една от тях.



Нека също предположим, че това е клетката, в която се намира A. Новият елемент изритва A от неговото място, A на свой ред отива на алтернативната си позиция и изритва B, от неговото място. Алтернативното място за B обаче е свободно, така че добавянето завършва успешно.

Да предположим, че клетката, от която се опитва да изрита елемент, новият елемент е тази, в която се намира H. Тогава се получава зацикляне тъй като H и W образуват цикъл. В този случай трябва да се изпълни пресъздаване на таблицата, използвайки нови хеш-функции и по-голям размер.

В най-опростената си версия този метод има константен достъп до елементите си и то в най-лошия случай, но това е изпълнено само при ограничението, че фактора на запълване е по-малък от 0.5.

Използването на три различни хеш-функции, вместо две може да доведе до ефективна горна граница на фактора на запълване до над 0.9.

Проучвания показват, че кукувичето хеширане и неговите варианти могат да бъдат много по-ефективни от широко използваните днес нареждане в списък и методите с отворено адресиране. Въпреки това все още този метод остава широко неизвестен и неизползван в практиката.

Структура от данни "множество"

В тази секция ще разгледаме абстрактната структура от данни множество (set) и две нейни типични реализации. Ще обясним предимствата и недостатъците им и в какви ситуации коя от имплементациите да предпочитаме.

Абстрактна структура данни "множество"

Множествата са колекции, в които няма повтарящи се елементи. В контекста на .NET това ще означава, че за всеки обект от множества извиквайки метода му **`Equals()`**, като подаваме като аргумент някои от другите обекти в множеството резултатът винаги ще е **`false`**.

Някои множества позволяват присъствието в себе си и на **`null`**, други не.

Освен, че не допуска повтарящи се обекти, друго важно нещо, което отличава множеството от списъците и масивите е, че неговите елементи си нямат номер. Елементите на множеството не могат да бъдат достъпвани по някакъв друг ключ, както е при речниците. Самите елементи играят ролята на ключ.

Единственият начин да достъпите обект от множество е като разполагате със самия обект или евентуално с обект, който е еквивалентен на него. Затова на практика достъпваме всички елементи на дадено множество наведнъж, докато го обхождаме в цикъл. Например чрез разширената конструкцията за **`for`** цикъл.

Основните операции, които се дефинират от структурата множество са следните:

- **`bool Add(element)`** – добавя в множеството зададен елемент, като ако вече има такъв елемент, връща **`false`**, а в противен случай **`true`**.
- **`bool Contains(element)`** – проверява дали множеството съдържа посочения елемент. Ако го има връща **`true`**, а в противен случай **`false`**.

- **`bool Remove(element)`** – премахва посочения елемент от множеството, ако съществува. Връща дали елементът е бил намерен.
- **`void Clear()`** – премахва всички елементи от множеството
- **`void IntersectWith(Set other)`** – в текущото множество остават само елементите от сечението на двете множества – това е множество, което съдържа всички елементи, които са едновременно и в едното и в другото множество.
- **`void UnionWith(Set other)`** – в текущото множество се натрупват елементите от обединението на двете множества – това е множество, което съдържа всички елементи, които са или в едното или в другото множество или и в двете.
- **`bool IsSubsetOf(Set other)`** – проверява дали текущото множество е подмножество на даденото множество. Връща **`true`** при положителен отговор и **`false`** при отрицателен
- **`bool IsSupersetOf(Set other)`** – проверява дали дадено множество е подмножество на текущото. Връща **`true`** при положителен отговор и **`false`** при отрицателен
- **`int Count`** – свойство което връща текущия брой на елементите в множеството

В .NET има една единствена публична имплементация на множество – това е класът **`HashSet<T>`** **`(assembly System.Core, namespace System.Collections.Generic)`**. Този клас имплементира множество чрез хеш-таблица. Но ако потърсим в асемблитата на .NET ще забележим, че има и вътрешен (**`internal`**) клас, който имплементира множество чрез червено-черно дърво – **`TreeSet`**. За съжаление **`TreeSet`** не може да бъде използван. Ако разгледаме внимателно имплементацията на тези класове ще видим, че те всъщност представляват речници, при които елементът е едновременно ключ и стойност на наредената двойка. Естествено, когато е удобно да работим с множества, трябва да ги предпочитаме, пред това да използваме речник. В .NET 4.0 вече има интерфейс **`ISet<T>`**.

Реализация с хеш-таблица – клас HashSet<T>

Както вече споменахме, реализацията на множество с хеш-таблица в .NET е класът **`HashSet<T>`**. Този клас, подобно на **`Dictionary<K, V>`**, има конструктори, чрез които може да се зададат списък с елементи, както и имплементация на **`IEqualityComparer`**, за който споменахме по-рано. Те имат същият смисъл, защото тук отново използваме хеш-таблица

Ето един пример, който демонстрира използване на множества и описаните в предния параграф основни операции - обединение и сечение:

```
class StudentListExample
```

```
{
    static void Main()
    {
        HashSet<string> aspNetStudents = new HashSet<string>();
        aspNetStudents.Add("S. Nakov");
        aspNetStudents.Add("V. Kolev");
        aspNetStudents.Add("M. Valkov");

        HashSet<string> silverlightStudents = new HashSet<string>();
        silverlightStudents.Add("S. Guthrie");
        silverlightStudents.Add("M. Valkov");

        HashSet<string> allStudents = new HashSet<string>();
        allStudents.UnionWith(aspNetStudents);
        allStudents.UnionWith(silverlightStudents);

        HashSet<string> intersectStudents =
            new HashSet<string>(aspNetStudents);
        intersectStudents.IntersectWith(silverlightStudents);

        Console.WriteLine("ASP.NET students: " +
            GetListOfStudents(aspNetStudents));
        Console.WriteLine("Silverlight students: " +
            GetListOfStudents(silverlightStudents));
        Console.WriteLine("All students: " +
            GetListOfStudents(allStudents));
        Console.WriteLine(
           "Students in both ASP.NET and Silverlight: " +
            GetListOfStudents(intersectStudents));
    }

    static string GetListOfStudents(IEnumerable<string> students)
    {
        string result = string.Empty;
        foreach (string student in students)
        {
            result += student + ", ";
        }

        if (result != string.Empty)
        {
            //remove the extra separator at the end
            result = result.TrimEnd(',', ' ');
        }

        return result;
    }
}
```

Резултатът от изпълнението е:

```
ASP.NET students: S. Nakov, V. Kolev, M. Valkov
Silverlight students: S. Guthrie, M. Valkov
All students: S. Nakov, V. Kolev, M. Valkov, S. Guthrie
Students in both ASP.NET and Silverlight: M. Valkov
```

Обърнете внимание, че **"M. Valkov"** присъства и в двете множества, но в обединението се появява само веднъж. Това е така, защото, както знаем, един елемент може да се съдържа най-много веднъж в дадено множество.

Реализация с черно-червено дърво – клас TreeSet<T>

TreeSet<T> представлява множество, реализирано чрез червено-черно дърво. В допълнение, то има свойството, че в него елементите се пазят подредени по големина. Това е причината в него да можем да добавяме само елементи, които са сравними. Припомняме, че в .NET това обикновено означава, че обектите са от клас, който имплементира **IComparable<T>**. Но както вече споменахме, в .NET класът **TreeSet<T>** е **internal** и следователно не можем да го ползваме. Или поне не директно. За щастие можем доста лесно да направим **TreeSet** с основните операции обединение и сечение, като ползваме **SortedDictionary<T>**. Ето и нашата реализация на **TreeSet**:

TreeSet.cs

```
using System;
using System.Collections.Generic;

/// <summary>
/// Class that represents ordered set, based on SortedDictionary
/// </summary>
/// <typeparam name="T">The type of the elements
/// in the set</typeparam>
public class TreeSet<T>: ICollection<T>
{
   private SortedDictionary<T, bool> innerDictionary;

   public TreeSet(IEnumerable<T> element): this()
   {
      foreach (T item in element)
      {
         this.Add(item);
      }
   }

   public TreeSet()
   {
      this.innerDictionary = new SortedDictionary<T, bool>();
```

```
}

/// <summary>
/// Adds an element to the set.
/// </summary>
/// <param name="element">element to add</param>
/// <returns>true if the element has not been
/// already added, false otherwise</returns>
public bool Add(T element)
{
   if (!innerDictionary.ContainsKey(element))
   {
      this.innerDictionary[element] = true;
      return true;
   }

   return false;
}

/// <summary>
/// Performs intersection of this set with the specified set
/// </summary>
/// <param name="other">Set to intersect with</param>
public void IntersectWith(TreeSet<T> other)
{
   List<T> elementsToRemove =
      new List<T>(Math.Min(this.Count, other.Count));

   foreach (T key in this.innerDictionary.Keys)
   {
      if (!other.Contains(key))
      {
         elementsToRemove.Add(key);
      }
   }

   foreach (T elementToRemove in elementsToRemove)
   {
      this.Remove(elementToRemove);
   }
}

/// <summary>
/// Performs an union operation with another set
/// </summary>
/// <param name="other">The set to perform union with</param>
public void UnionWith(TreeSet<T> other)
{
   foreach (T key in other)
   {
    this.innerDictionary[key] = true;
  }
}

#region ICollection<T> Members

void ICollection<T>.Add(T item)
{
  this.Add(item);
}

public void Clear()
{
  this.innerDictionary.Clear();
}

public bool Contains(T item)
{
  return this.innerDictionary.ContainsKey(item);
}

public void CopyTo(T[] array, int arrayIndex)
{
  this.innerDictionary.Keys.CopyTo(array, arrayIndex);
}

public int Count
{
  get { return this.innerDictionary.Count; }
}

public bool IsReadOnly
{
  get { return false; }
}

public bool Remove(T item)
{
  return this.innerDictionary.Remove(item);
}

#endregion

#region IEnumerable<T> Members

public IEnumerator<T> GetEnumerator()
{
  return this.innerDictionary.Keys.GetEnumerator();
}
```

```
  #endregion

  #region IEnumerable Members

  System.Collections.IEnumerator
    System.Collections.IEnumerable.GetEnumerator()
  {
    return innerDictionary.Keys.GetEnumerator();
  }

  #endregion
}
```

Ще демонстрираме работата с класа **`TreeSet<T>`** с един пример:

```
class Program
{
  static void Main()
  {
    TreeSet<string> bandsIvanchoLikes = new TreeSet<string>(
    new[] {
  "manowar", "blind guardian", "dio",  "grave digger", "kiss",
  "dream theater", "manowar", "megadeth", "dream theater",
  "dio", "judas priest", "manowar", "kreator",
  "blind guardian", "iron maiden", "accept", "dream theater"
    });

    TreeSet<string> bandsMariikaLikes = new TreeSet<string>(
    new[] {
  "iron maiden", "dio", "accept", "manowar", "slayer",
  "megadeth", "dio", "manowar", "running wild", "grave digger",
  "accept", "kiss", "manowar", "iron maiden", "manowar",
  "judas priest", "iced earth", "manowar", "dio",
  "iron maiden", "manowar", "slayer"
    });

    Console.WriteLine("Ivancho likes these bands: ");
    Console.WriteLine(
      GetCommaSeparatedList(bandsIvanchoLikes));
    Console.WriteLine();

    Console.WriteLine("Mariika likes these bands: ");
    Console.WriteLine(
      GetCommaSeparatedList(bandsMariikaLikes));
    Console.WriteLine();

    TreeSet<string> intersectBands = new TreeSet<string>();
    intersectBands.UnionWith(bandsIvanchoLikes);
    intersectBands.IntersectWith(bandsMariikaLikes);
    Console.WriteLine(string.Format(
      "Does Ivancho like Mariika? {0}",
      intersectBands.Count > 5 ? "Yes!" : "No!"));

    Console.WriteLine(
      "Because Ivancho and Mariika both like: ");
    Console.WriteLine(
      GetCommaSeparatedList(intersectBands));
    Console.WriteLine();

    TreeSet<string> uniounBands = new TreeSet<string>();
    uniounBands.UnionWith(bandsIvanchoLikes);
    uniounBands.UnionWith(bandsMariikaLikes);

    Console.WriteLine(
      "All bands that Ivancho or Mariika like:");
    Console.WriteLine(
      GetCommaSeparatedList(uniounBands));
  }

  static string GetCommaSeparatedList(
    IEnumerable<string> items)
  {
    string result = string.Empty;

    int i = 1;
    foreach (string item in items)
    {
      result += item + ", ";
      if (i % 3 == 0) //3 elements on a line
      {
        result += Environment.NewLine;
      }

      i++;
    }

    if (result != string.Empty)
    {
      //remove the extra separators at the end
      result = result.TrimEnd(',', ' ', '\r', '\n');
    }

    return result;
  }
}
```

След изпълнението на програмата получаваме следния резултат:

```
Ivancho likes these bands:
```

```
accept, blind guardian, dio,
dream theater, grave digger, iron maiden,
judas priest, kiss, kreator,
manowar, megadeth

Mariika likes these bands:
accept, dio, grave digger,
iced earth, iron maiden, judas priest,
kiss, manowar, megadeth,
running wild, slayer

Do Ivancho and Mariika like each other? Yes!
Because Ivancho and Mariika both like:
accept, dio, grave digger,
iron maiden, judas priest, kiss,
manowar, megadeth

All bands that Ivancho or Mariika like:
accept, blind guardian, dio,
dream theater, grave digger, iced earth,
iron maiden, judas priest, kiss,
kreator, manowar, megadeth,
running wild, slayer
```

Това, което можем веднага да забележим е, че елементите в нашето множество, за разлика от **HashSet** са винаги подредени.

В .NET Framework версия 4.0 вече има клас **SortedSet<T>** и интерфейс **ISet<T>**. Можете да се запознаете с тяхната имплементация, използвайки някой от декомпилаторите, описани в секцията "Декомпилиране на код".

За читателя остава задачата, да разшири функционалността на множеството с други операции. Това, за което е важно да си дадем сметка, е че работата с множества е наистина лесна и проста. Ако познаваме добре тяхната структура и свойства, ще можем да ги използвате ефективно и на място.

Упражнения

1. Напишете програма, която брои колко пъти се среща всяко число в дадена редица от числа.

   Пример: `array = {3, 4, 4, 2, 3, 3, 4, 3, 2}`

   2 → 2 пъти

   3 → 4 пъти

   4 → 3 пъти

2. Напишете програма, която премахва всички числа, които се срещат нечетен брой пъти в дадена редица. Например, ако имаме началната

редица `{4, 2, 2, 5, 2, 3, 2, 3, 1, 5, 2, 6, 6, 6}`, трябва да я редуцираме до редицата `{5, 3, 3, 5}`.

3. Напишете програма, която по даден текст във текстов файл, преброява колко пъти се среща всяка дума. Отпечатайте на конзолата всички думи и по колко пъти се срещат, подредени по брой срещания.

   Пример: "`This is the TEXT. Text, text, text – THIS TEXT! Is this the text?`"

   Резултат:

   `is → 2, the → 2, this → 3, text → 6`

4. Реализирайте клас **`DictHashSet<T>`**, базиран на класа **`HashDictionary<K, V>`**, който разгледахме по-горе.
5. Реализирайте хеш-таблица, която съхранява тройки стойности (ключ1, ключ2, стойност) и позволява бързо търсене по двойка ключове и добавяне на тройки стойности.
6. Реализирайте хеш-таблица, която позволява по даден ключ да съхраняваме повече от една стойност.
7. Реализирайте хеш-таблица, която използва "кукувиче хеширане" с 3 хеш-функции за разрешаване на колизиите.
8. Реализирайте структурата данни хеш-таблица в клас **`HashTable<K, T>`**. Пазете данните в масив от списъци от двойки ключ-стойност (**`LinkedList<KeyValuePair<K,T>>[]`**) с начален капацитет от 16 елемента. Когато хеш-таблицата достигне 75% от своя капацитет да се удвоява капацитета. Реализирайте следните операции: **`Add(key, value)`**, **`Find(key)→value`**, **`Remove(key)`**, **`Count`**, **`Clear()`**, **`this[]`**, **`Keys`**. Реализирайте и итериране по елементите на хеш-таблицата с **`foreach`**.
9. Реализирайте структурата от данни "**`Set`**" в клас **`HashedSet<T>`**. Използвайте класа от предната задача **`HashTable<K, T>`**, за да пазите елементите. Имплементирайте всички стандартни операции за типа данни **`Set`**: **`Add(T)`**, **`Find(T)`**, **`Remove(T)`**, **`Count`**, **`Clear()`**, обединение и сечение.
10. Дадени са три редици от числа, дефинирани чрез формулите:

    - `f1(0) = 1; f1(k) = 2*f1(k-1) + 3;  f1 = {1, 5, 13, 29, …}`
    - `f2(0) = 2; f2(k) = 3*f2(k-1) + 1;  f2 = {2, 7, 22, 67, …}`
    - `f3(0) = 2; f3(k) = 2*f3(k-1) - 1;  f3 = {2, 3, 5, 9, …}`

    Напишете програма, която намира сечението и обединението на множествата от членовете на редиците в интервала [0; 100000]: $f_1 * f_2$; $f_1 * f_3$; $f_2 * f_3$; $f_1 * f_2 * f_3$; $f_1 + f_2$; $f_1 + f_3$; $f_2 + f_3$; $f_1 + f_2 + f_3$. Със символите + и * означаваме съответно обединение и сечение на множества.

11. * Дефинирайте клас `TreeMultiSet<T>`, който позволява да пазим съвкупност от елементи, подредени по големина и позволява повторения на някои от елементите. Реализирайте операциите добавяне на елемент, търсене на броя срещания на даден елемент, изтриване на елемент, итератор, намиране на най-малък / най-голям елемент, изтриване на най-малък / най-голям елемент. Реализирайте възможност за подаване на външен `Comparer<T>` за сравнение на елементите.

12. * Даден е списък с времената на пристигане и заминаване на всички автобуси от дадена автогара. Да се напише програма, която използвайки `HashSet` класа по даден интервал (начало, край) намира броя автобуси, които успяват да пристигнат и да напуснат автогарата. Пример:

    Имаме данните за следните автобуси: [08:24-08:33], [08:20-09:00], [08:32-08:37], [09:00-09:15]. Даден е интервалът [08:22-09:05]. Броят автобуси, които идват и си тръгват в рамките на този интервал е 2.

13. * Дадена е редица `P` с цели числа (1 < P < 50 000) и число N. Щастлива под-редица в редицата `P` наричаме всяка съвкупност, състояща се от последователни числа от `P`, чиято сума е N. Да си представим, че имаме редицата `S`, състояща се от всички щастливи под-редици в `P`, подредени в намаляващ ред спрямо дължината им. Напишете програма, която извежда първите 10 елемента на `S`.

    Пример: Имаме N=5 и редицата `P={1, 1, 2, 1, -1, 2, 3, -1, 1, 2, 3, 5, 1, -1, 2, 3}`. Редицата `S` се състои от следните 13 под-редици на `P`:

    - **[1, -1, 2, 3, -1, 1]**
    - **[1, 2, 1, -1, 2]**
    - **[1, -1, 2, 3]**
    - **[2, 3, -1, 1]**
    - **[3, -1, 1, 2]**
    - **[-1, 1, 2, 3]**
    - **[1, -1, 2, 3]**
    - **[1, 1, 2, 1]**
    - **[5, 1, -1]**
    - **[2, 3]**
    - [2, 3]
    - [2, 3]
    - [5]

Първите 10 елемента на P са дадени с удебелен шрифт.

Решения и упътвания

1. Използвайте **`Dictionary<TKey,TValue>`**
2. Използвайте **`Dictionary`** и **`ArrayList`**.
3. Използвайте **`Dictionary`** с ключ дума и стойност – броя срещания. След като преброите всички думи, сортирате речника по стойност.
4. Използвайте за ключ и за стойност една и съща стойност – елементът от множеството.
5. Използвайте хеш-таблица от хеш-таблици.
6. Ползвайте **`Dictionary<K, ArrayList<V>>`**.
7. Можете за първа хеш-функция да ползвате **`GetHashCode() % size`**, за втора да ползвате **`(GetHashCode () * 83 + 7) % size`**, а за трета – (**`GetHashCode () * GetHashCode () + 19) % size`**).
8. За да удвоите размера на вашата колекция, можете да заделите двойно по-голям масив и да прехвърлите елементите от стария в новия, след което да насочите референцията от стария масив към новия. За да имплементирате **`foreach`** оператора върху вашата колекция, имплементирайте интерфейса **`IEnumerable`** и във вашия метод **`GetEnumerator()`** да връщате съответния метод **`GetEnumerator()`** на масива от списъци. Можете да използвате и оператора **`yield`**.
9. Един вариант да решите задачата е, да използвате за ключ в хеш-таблицата елемента от множеството, а за стойност винаги **`true`**. Обединението и сечението ще извършвате с изцикляне по елементите на едното множество и проверка дали в едното множество има (съответно няма) елемента от другото множество.
10. Намерете всички членове на трите редици в посочения интервал и след това използвайки **`HashSet<int>`** реализирайте обединение и сечение на множества, след което направете исканите пресмятания.
11. Класът **`TreeMultiSet<T>`** можете да реализираме чрез **`SortedDictionary<K, int>`**, който пази броя срещания на всеки от ключовете.
12. Очевидното решение е да проверим всеки от автобусите дали пристига и си тръгва в посочения интервал. Според условието на задачата, обаче, трябва да ползваме класа **`HashSet`**.

    Решението е такова: Можем да намерим множествата на всички автобуси, които пристигат след началния час и на всички автобуси, отпътуващи преди крайния час. Сечението на тези множества дава търсените автобуси. Ако **`TimeInterval`** е клас който съхранява разписанието на един автобус, сечението можем да намерим с **`HashSet< TimeInterval>`** при подходящо дефинирани **`GetHashCode()`** и **`Equals()`**.

13. Първата идея за решаване на задачата е проста: с два вложени цикъла намираме всички щастливи под-редици на редицата P, след което ги сортираме по дължината им и накрая извеждаме първите 10. Това, обаче няма да работи добре, ако броят щастливи под-редици са десетки милиони.

    Ще опишем една идея за по-ефективно решение. Ще използваме класа **`TreeMultiSet<T>`**. В него ще съхраняваме първите 10 под-редици от **`S`**, т.е. мулти-множество от щастливите под-редици на **`P`**, подредени по дължина в намаляващ ред. Когато имаме 10 под-редици в мулти-множеството и добавим нова 11-та под-редица, тя ще застане на мястото си заради Comparer-а, който сме дефинирали. След това можем веднага да изтрием последната под-редица от мулти-множеството, защото тя не е сред първите 10. Така във всеки един момент ще пазим текущите 10 най-дълги под-редици. По този начин ще консумираме много по-малко памет и ще избегнем сортирането накрая. Имплементацията няма да е лесна, така че отделете достатъчно време!

Глава 19. Структури от данни – съпоставка и препоръки

В тази тема...

В настоящата тема ще съпоставим една с друга структурите данни, които разгледахме до момента, по отношение на скоростта, с която извършват основните операции (добавяне, търсене, изтриване и т.н.). Ще дадем конкретни препоръки в какви ситуации какви структури от данни да ползваме. Ще обясним кога да предпочетем хеш-таблица, кога масив, кога динамичен масив, кога множество, реализирано чрез хеш-таблица и кога балансирано дърво. Почти всички тези структури имат вградена имплементация в .NET Framework. От нас се очаква единствено да можем да преценяваме кога коя структура да ползваме, за да пишем ефективен и надежден програмен код.

Защо са толкова важни структурите данни?

Може би се чудите защо отделяме толкова голямо внимание на структурите данни и защо ги разглеждаме в такива големи детайли? Причината е, че сме си поставили за задача да ви направим мислещи софтуерни инженери. Без да познавате добре основните структури от данни в програмирането и основните компютърни алгоритми, вие не можете да бъдете добри програмисти и рискувате да си останете обикновени "занаятчии". Който владее добре структурите от данни и алгоритми и успее да си развие мисленето в посока правилното им използване, има големи шансове да стане добър софтуерен инженер – който анализира проблемите в дълбочина и предлага ефективни решения.

По темата защо са важни структурите от данни и алгоритмите има изписани стотици книги. Особено впечатляващи са четирите тома на Доналд Кнут, озаглавени "The Art of Computer Programming", в които структурите от данни и алгоритмите са разгледани в над 2500 страници. Един автор дори е озаглавил книга с отговора на въпроса "защо структурите от данни са толкова важни". Това е книгата на Никлаус Вирт "Алгоритми + структури от данни = програми", в която се разглеждат отново структурите данни и фундаменталните алгоритми в програмирането.

Структурите от данни и алгоритмите стоят в основата на програмирането. За да станете добри програмисти, е необходимо да познавате основните структури от данни и алгоритми и да се научите да ги прилагате по подходящ начин.

В много голяма степен и нашата книга е насочена именно към изучаването на основните структури от данни и алгоритми в програмирането, като сме се стремили да ги илюстрираме в контекста на съвременното софтуерно инженерство с .NET платформата.

Сложност на алгоритъм

Не може да се говори за ефективност на алгоритми и структури от данни, без да се използва понятието "сложност на алгоритъм", с което вече се сблъскахме няколко пъти под една или друга форма. Няма да даваме математическа дефиниция, за да не натоварваме читателите, а ще дадем неформално обяснение.

Сложност на алгоритъм е мярка, която отразява порядъка на броя операции, необходими за изпълнение на дадена операция или алгоритъм като функция на обема на входните данни. Формулирано още по-просто, сложност е груба, приблизителна оценка на броя стъпки за изпълнение на даден алгоритъм. При оценяването на сложност говорим за порядъка на броя операции, а не за техния точен брой. Например ако имаме от порядъка на N^2 операции за обработката на N елемента, то $N^2/2$ и $3* N^2$ са брой операции от един и същ квадратичен порядък. Сложността на алгоритмите се означава най-често с нотацията O(f), където f е функция на размера (обема) на входните данни.

Сложността може да бъде константна, логаритмична, линейна, n*log(n), квадратична, кубична, експоненциална и друга. Това означава, че се изпълняват съответно от порядъка на константен, логаритмичен, линеен и т.н. брой стъпки за решаването на даден проблем.

Сложност на алгоритъм е груба оценка на броя стъпки, които алгоритъмът ще направи в зависимост от размера на входните данни. Това е груба оценка, която се интересува от порядъка на броя стъпки, а не от точния им брой.

Типични сложности на алгоритмите

Ще обясним какво означават видовете сложност чрез следната таблица:

Сложност	Означение	Описание
константна	O(1)	За извършване на дадена операция са необходими константен брой стъпки (например 1, 5, 10 или друго число) и този брой не зависи от обема на входните данни.
логаритмична	O(log(N))	За извършване на дадена операция върху N елемента са необходими брой стъпки от порядъка на log(N), където основата на логаритъма е най-често 2. Например алгоритъм със сложност O(log(N)) за N = 1 000 000 ще направи около 20 стъпки (с точност до константа). Тъй като основата на логаритъма няма съществено значение за порядъка на броя операции, тя обикновено се изпуска.
линейна	O(N)	За извършване на дадена операция върху N елемента са необходими приблизително толкова стъпки, колкото са елементите. Например за 1 000 елемента са нужни около 1 000 стъпки. Линейната сложност означава, че броят елементи и броят операции са линейно зависими, например броят стъпки за N елемента е около N/2 или 3*N.

	$O(n*\log(n))$	За извършване на дадена операция върху N елемента са необходими приблизително N*log(N) стъпки. Например при 1 000 елемента са нужни около 10 000 стъпки.
квадратична	$O(n^2)$	За извършване на дадена операция са необходими от порядъка на N^2 на брой стъпки, където N характеризира обема на входните данни. Например за дадена операция върху 100 елемента са необходими 10 000 стъпки. Реално квадратична сложност имаме, когато броят стъпки е в квадратна зависимост спрямо обема на входните данни, например за N елемента стъпките могат да са от порядъка на $3*N^2/2$.
кубична	$O(n^3)$	За извършване на дадена операция са необходими от порядъка на N^3 стъпки, където N характеризира обема на входните данни. Например при 100 елемента се изпълняват около 1 000 000 стъпки.
експоненциална	$O(2^n)$, O(N!), $O(n^k)$, …	За извършване на дадена операция или изчисление са необходими брой стъпки, който е в експоненциална зависимост спрямо размера на входните данни. Например при N=10 експоненциалната функция 2^N има стойност 1024, при N=20 има стойност 1 048 576, а при N=100 функцията има стойност, която е число с около 30 цифри. Експоненциалната функция N! расте още по-бързо: за N=5 има стойност 120, за N=10 има стойност 3 628 800, а за N=20 – 2 432 902 008 176 640 000.

При оценката на сложност константите не се взимат предвид, тъй като не влияят съществено на броя операции. По тази причина алгоритъм, който извършва N стъпки и алгоритми, които извършват съответно N/2 и 3*N стъпки се считат за линейни и за приблизително еднакво ефективни, тъй като извършват брой операции, които са от един и същ порядък.

Сложност и време за изпълнение

Скоростта на изпълнение на програмата е в пряка зависимост от сложността на алгоритъма, който се изпълнява. Ако тази сложност е малка, програмата ще работи бързо, дори за голям брой елементи. Ако сложността е голяма, програмата ще работи бавно или въобще няма да работи (т.е. ще заспи) при голям брой елементи.

Ако вземем един средностатистически компютър от 2008 година, можем да приемем, че той изпълнява около 50 000 000 елементарни операции в секунда. Разбира се, това число трябва да ви служи единствено за груб ориентир. Различните процесори работят с различна скорост и различните елементарни операции се изпълняват с различна скорост, а и компютърната техника постоянно напредва. Все пак, ако приемем, че използваме средностатистически домашен компютър от 2008 г., можем да направим следните изводи за скоростта на изпълнение на дадена програма в зависимост от сложността на алгоритъма и обема на входните данни:

алгоритъм	**10**	**20**	**50**	**100**	**1 000**	**10 000**	**100 000**
O(1)	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.
O(log(n))	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.
O(n)	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.
O(n*log(n))	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.
O(n^2)	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	2 сек.	3-4 мин.
O(n^3)	< 1 сек.	< 1 сек.	< 1 сек.	< 1 сек.	20 сек.	5.55 часа	231.5 дни
O(2^n)	< 1 сек.	< 1 сек.	260 дни	заспива	заспива	заспива	заспива
O(n!)	< 1 сек.	заспива	заспива	заспива	заспива	заспива	заспива
O(n^n)	3-4 мин.	заспива	заспива	заспива	заспива	заспива	заспива

От таблицата можем да направим много изводи:

- Алгоритми с константна, логаритмична и линейна сложност са толкова бързи, че не можем да усетим забавяне, дори при относително голям размер на входните данни.
- Сложността O(n*log(n)) е близка до линейната и също работи толкова, бързо, че трудно можем да усетим забавяне.
- Квадратични алгоритми работят добре до няколко хиляди елемента.
- Кубични алгоритми работят добре при под 1 000 елемента.

- Като цяло т.нар. полиномиални алгоритми (тези, които не са експоненциални) се считат за бързи и работят добре за хиляди елементи.
- Експоненциалните алгоритми като цяло не работят и трябва да ги избягваме (когато е възможно). Ако имаме експоненциално решение за дадена задача, може да се каже, че всъщност нямаме решение, защото то ще работи само ако елементите са под 10-20. Съвременната криптография разчита точно на това – че не са известни бързи (неекспоненциални) алгоритми за откриване на тайните ключове, които се използват за шифриране на данните.

Ако решите една задача с експоненциална сложност, това означава, че сте я решили само за много малък размер на входните данни и в общия случай решението ви не работи.

Разбира се, данните в таблицата са само ориентировъчни. Понякога може да се случи линеен алгоритъм да работи по-бавно от квадратичен или квадратичен да работи по-добре от O(n*log(n)). Причините за това могат да са много:

- Възможно е константите за алгоритъм с малка сложност да са големи и това да направи алгоритъма бавен като цяло. Например, ако имаме алгоритъм, който прави 50*n стъпки и друг, който прави 1/100*n*n стъпки, то за стойности до 5000 квадратичният алгоритъм е по-бърз от линейния.
- Понеже оценката на сложността се прави за най-лошия случай, е възможно квадратичен алгоритъм да работи по-добре от алгоритъм O(n*log(n)) в 99% от случаите. Можем да дадем пример с алгоритъма `QuickSort` (стандартния за .NET Framework сортиращ алгоритъм), който в средния случай работи малко по-добре от `MergeSort` (сортиране чрез сливане), но в най-лошия случай `QuickSort` прави от порядъка на n^2 стъпки, докато `MergeSort` прави винаги O(n*log(n)) стъпки.
- Възможно е алгоритъм, който е оценен, че работи с линейна сложност, да не работи толкова бързо, колкото се очаква заради неточна оценка на сложността. Например, ако търсим дадена дума в масив от думи, сложността е линейна, но на всяка стъпка се извършва сравнение на символни низове, което не е елементарна операция и може да отнеме много повече време, отколкото извършването на една елементарна операция (например сравнение на два символни низа).

Сложност по няколко променливи

Сложността може да зависи и от няколко входни променливи едновременно. Например, ако търсим елемент в правоъгълна матрица с размери M на N, то скоростта на търсенето зависи и от M и от N. Понеже в най-лошия случай трябва да обходим цялата матрица, то ще направим най-много M*N на брой стъпки. Така сложността се оценява като O(M*N).

Най-добър, най-лош и среден случай

Сложността на алгоритмите се оценява обикновено в най-лошия случай (при най-неблагоприятния сценарий). Това означава, че в средния случай те могат да работят и по-бързо, но в най-лошия случай работят с посочената сложност и не по-бавно.

Да вземем един пример: търсене на елемент в масив по даден ключ. За да намерим търсения ключ, трябва да проверим в най-лошия случай всички елементи на масива. В най-добрия случай ще имаме късмет и ще намерим търсения ключ още в първия елемент. В средния случай можем да очакваме да проверим средно половината елементи на масива докато намерим търсения. Следователно в най-лошия случай сложността е O(N), т.е. линейна. В средния случай сложността е O(N/2) = O(N), т.е. отново линейна, защото при оценяване на сложност константите се пренебрегват. В най-добрия случай имаме константна сложност O(1), защото изпълняваме само една стъпка и с нея директно откриваме търсения елемент.

Приблизително оценена сложност

Понякога е трудно да оценим точно сложността на даден алгоритъм, тъй като изпълняваме операции, за които не знаем точно колко време отнемат и колко стъпки изпълняват вътрешно. Да вземем за пример търсенето на дадена дума в масив от символни низове (текстове). Задачата е лесна: трябва да обходим масива и във всеки от текстовете да търсим със **`Substring()`** или с регулярен израз дадената дума. Можем да си зададем въпроса: ако имаме 10 000 текста, това бързо ли ще работи? А какво ще стане ако текстовете са 100 000? Ако помислим внимателно, ще установим, че за да оценим адекватно скоростта на търсенето, трябва да знаем колко са обемни текстовете, защото има разлика между търсене в имена на хора (които са до около 100 символа) и търсене в научни статии (които са съставени от средно 20 000 – 30 000 символа). Все пак можем да оценим сложността спрямо обема на текстовете, в които търсим: тя е най-малко O(L), където L е сумата от дължините на всички текстове. Това е доста груба оценка, но е много по-точна, отколкото да кажем, че сложността е O(N), където N е броят текстове, нали? Трябва да помислим дали взимаме предвид всички ситуации, които биха могли да възникнат. Има ли значение колко дълга дума търсим в масива от текстове? Вероятно търсенето на дълги думи работи по-бавно от търсенето на кратки думи. Всъщност нещата стоят малко по-различно. Ако търсим "aaaaaaa" в текста "aaaaaabaaaaacaaaaaabaaaaacaaaaab", това ще е по-бавно, отколкото ако търсим "xxx" в същия текст, защото в първия случай ще имаме много повече поредици съвпадения, отколкото във втория. Следователно при някои специални ситуации, търсенето зависи съществено и от дължината на търсената дума и оценката O(L) може да се окаже силно занижена.

Сложност по памет

Освен броя стъпки чрез функция на входните данни могат да се измерват и други ресурси, които алгоритъма използва, например памет, брой дискови операции и т.н. За някои алгоритми скоростта на изпълнение не е толкова важна, колкото обема на паметта, която ползват. Например, ако един алгоритъм е линеен, но използва оперативна памет от порядъка на N^2, той вероятно ще страда от недостиг на памет при N=100 000 (тогава ще му трябват от порядъка на 9 GB оперативна памет), въпреки, че би следвало да работи много бързо.

Оценяване на сложност – примери

Ще дадем няколко примера, с които ще ви покажем как можете да оценявате сложността на вашите алгоритми и да преценявате дали ще работи бързо написаният от вас програмен код:

Ако имаме единичен цикъл от 1 до N, сложността му е линейна – O(N):

```
int FindMaxElement(int[] array)
{
   int max = int.MinValue;
   for (int i = 1; i < array.Length; i++)
   {
      if (array[i] > max)
      {
         max = array[i];
      }
   }
   return max;
}
```

Този код ще работи добре, дори при голям брой елементи.

Ако имаме два вложени цикъла от 1 до N, сложността им е квадратична – $O(N^2)$. Пример:

```
int FindInversions(int[] array)
{
   int inversions = 0;
   for (int i = 0; i < array.Length - 1; i++)
   {
      for (int j = i + 1; j < array.Length; j++)
      {
         if (array[i] > array[j])
         {
            inversions++;
         }
      }
   }
   return inversions;
}
```

Този код ще работи добре, ако елементите не са повече от няколко хиляди или десетки хиляди.

Ако имаме три вложени цикъла от 1 до N, сложността им е кубична – O(N^3). Пример:

```
long Sum3(int n)
{
   long sum = 0;
   for (int a = 1; a < n; a++)
   {
      for (int b = 1; b < n; b++)
      {
         for (int c = 1; c < n; c++)
         {
            sum += a * b * c;
         }
      }
   }
   return sum;
}
```

Този код ще работи добре, ако елементите в масива са под 1 000.

Ако имаме два вложени цикъла съответно от 1 до N и от 1 до M, сложността им е квадратична – O(N*M). Пример:

```
long SumMN(int n, int m)
{
   long sum = 0;
   for (int x = 1; x <= n; x++)
   {
      for (int y = 1; y <= m; y++)
      {
         sum += x * y;
      }
   }
   return sum;
}
```

Скоростта на този код зависи от две променливи. Кодът ще работи добре, ако M, N < 10 000 или ако поне едната променлива има достатъчно малка стойност.

Трябва да обърнем внимание на факта, че не винаги три вложени цикъла означават кубична сложност. Ето един пример, при който сложността е O(N*M):

```
long SumMN(int n, int m)
{
   long sum = 0;
   for (int x = 1; x <= n; x++)
   {
      for (int y = 1; y <= m; y++)
      {
         if (x == y)
         {
            for (int i = 1; i <= n; i++)
            {
               sum += i * x * y;
            }
         }
      }
   }
   return sum;
}
```

В този пример най-вътрешният цикъл се изпълнява точно min(M, N) пъти и не оказва съществено влияние върху скоростта на алгоритъма. Горният код изпълнява приблизително N*M + min(M,N)*N стъпки, т.е. сложността му е квадратична.

При използване на рекурсия сложността е по-трудно да се определи. Ето един пример:

```
long Factorial(int n)
{
   if (n == 0)
   {
      return 1;
   }
   else
   {
      return n * Factorial(n - 1);
   }
}
```

В този пример сложността е очевидно линейна – O(N), защото функцията **`factorial()`** се изпълнява точно веднъж за всяко от числата 1, 2, ..., n.

Ето една рекурсивна функция, за която е много по-трудно да се сметне сложността:

```
long Fibonacci(int n)
{
   if (n == 0)
   {
      return 1;
```

```
    }
    else if (n == 1)
    {
        return 1;
    }
    else
    {
        return Fibonacci(n - 1) + Fibonacci(n - 2);
    }
}
```

Ако разпишем какво се случва при изпълнението на горния код, ще установим, че функцията се извиква толкова пъти, колкото е числото на Фибоначи с номер n+1. Можем грубо да оценим сложността и по друг начин: понеже на всяка стъпка от изпълнението на функцията се извършват средно по 2 рекурсивни извиквания, то броят рекурсивни извиквания би трябвало да е от порядъка на 2^n, т.е. имаме експоненциална сложност. Това автоматично означава, че са стойности над 20-30 функцията "ще зависне".

Същата функция за изчисление на n-тото число на Фибоначи можем да напишем с линейна сложност по следния начин:

```
long Fibonacci(int n)
{
    long fn = 1;
    long fn1 = 1;
    long fn2 = 1;
    for (int i = 2; i < n; i++)
    {
        fn = fn1 + fn2;
        fn2 = fn1;
        fn1 = fn;
    }
    return fn;
}
```

Виждате, че оценката на сложността ни помага да предвидим, че даден код ще работи бавно, още преди да сме го изпълнили и ни подсказва, че трябва да търсим по-ефективно решение.

Сравнение на основните структури от данни

След като се запознахме с понятието сложност на алгоритъм, вече сме готови да направим съпоставка на основните структури от данни, които разгледахме до момента, и да оценим с каква сложност всяка от тях извършва основните операции като добавяне, търсене, изтриване и други. Така ще можем лесно да съобразяваме според операциите, които са ни необходими, коя структура от данни ще е най-подходяща. В таблицата по-долу са дадени сложностите на основните операции при основните структури данни, които разгледахме в предходните глави:

структура	добавяне	търсене	изтриване	достъп по индекс
масив (`T[]`)	O(N)	O(N)	O(N)	O(1)
свързан списък (`LinkedList<T>`)	O(1)	O(N)	O(N)	O(N)
динамичен масив (`List<T>`)	O(1)	O(N)	O(N)	O(1)
стек (`Stack<T>`)	O(1)	-	O(1)	-
опашка (`Queue<T>`)	O(1)	-	O(1)	-
речник реализиран с хеш-таблица (`Dictionary<K, T>`)	O(1)	O(1)	O(1)	-
речник реализиран с балансирано дърво (`SortedDictionary<K, T>`)	O(log(N))	O(log(N))	O(log(N))	-
множество реализирано с хеш-таблица (`HashSet<T>`)	O(1)	O(1)	O(1)	-
множество реализирано с балансирано дърво (`SortedSet<T>`)	O(log(N))	O(log(N))	O(log(N))	-

Оставяме на читателя да помисли как точно се получават тези сложности.

Кога да използваме дадена структура?

Нека разгледаме всяка от посочените в таблицата структури от данни поотделно и обясним в какви ситуации е подходящо да се ползва такава структура и как се получават сложностите, дадени в таблицата.

Масив (T[])

Масивите са наредени съвкупности от фиксиран брой елементи от даден тип (например числа), до които достъпът става по индекс. Масивите представляват област от паметта с определен, предварително зададен размер. Добавянето на нов елемент в масив е много бавна операция, защото реално трябва да се задели нов масив с размерност по-голяма с 1 от текущата и да се прехвърлят старите елементи в новия масив. Търсенето в масив изисква сравнение на всеки елемент с търсената стойност. В средния случай са необходими N/2 сравнения. Изтриването от масив е много бавна операция, защото е свързана със заделяне на масив с размер с 1 по-малък от текущия и преместване на всички елементи без изтрития в новия масив. Достъпът по индекс става директно и затова е много бърза операция.

Масивите трябва да се ползват само когато трябва да обработим фиксиран брой елементи, до които е необходим достъп по индекс. Например, ако сортираме числа, можем да запишем числата в масив и да приложим някой от добре известните алгоритми за сортиране. Когато по време на работа е необходимо да променяме броя елементи, с които работим, масивът не е подходяща структура от данни.

Използвайте масиви, когато трябва да обработите фиксиран брой елементи, до които ви е необходим достъп по индекс.

Свързан / двусвързан списък (LinkedList<T>)

Свързаният списък и неговият вариант двусвързан списък съхраняват наредена съвкупност от елементи. Добавянето е бърза операция, но е малко по-бавна от добавяне в `List<T>`, защото всяко добавяне заделя памет. Заделянето на памет работи със скорост, която трудно може да бъде предвидена. Търсенето в свързан списък е бавна операция, защото е свързано с обхождане на всички негови елементи. Достъпът до елемент по индекс е бавна операция, защото в свързания списък няма индексиране и се налага обхождане на списъка, започвайки от началния елемент и придвижвайки се напред елемент по елемент. Изтриването на елемент по индекс е бавна операция, защото достигането до елемента с посочения индекс е бавна операция. Изтриването по стойност на елемент също е бавно, защото включва в себе си търсене.

Свързаният списък може бързо (с константна сложност) да добавя и изтрива елементи от двата си края, поради което е удобен за имплементация на стекове, опашки и други подобни структури.

Свързан списък в практиката се използва много рядко, защото динамично-разширяемият масив (`List<T>`) изпълнява почти всички операции, които могат да бъдат изпълнени с `LinkedList`, но за повечето от тях работи по-бързо и по-удобно.

Ползвайте `List<T>`, когато ви трябва свързан списък – той работи не по-бавно, а ви дава по-голяма бързина и удобство. Ползвайте `LinkedList`, ако има нужда от добавяне и изтриване на елементи в двата края на структурата.

Използвайте свързан списък (`LinkedList<T>`), когато трябва да добавяте и изтривате елементи от двата края на списъка. В противен случай ползвайте `List<T>`.

Динамичен масив (List<T>)

Динамичният масив (`List<T>`) е една от най-използваните в практиката структура от данни. Той няма фиксиран размер, както масивите, и позволява директен достъп по индекс, за разлика от свързания списък (`LinkedList<T>`). Динамичният масив е известен още с наименованията "списък реализиран с масив" и "динамично-разширяем масив".

`List<T>` вътрешно съхранява елементите си в масив, който има размер по-голям от броя съхранени елементи. При добавяне на елемент обикновено във вътрешния масив има свободно място и затова тази операция отнема константно време. Понякога масивът се препълва и се налага да се разшири. Това отнема линейно време, но се случва много рядко. В крайна сметка при голям брой добавяния усреднената сложност на добавянето на елемент към `List<T>` е константна – O(1). Тази усреднена сложност се нарича **амортизирана сложност**. Амортизирана линейна сложност означава, че ако добавим последователно 10 000 елемента, ще извършим сумарно брой стъпки от порядъка на 10 000 и болшинството от тях ще се изпълнят за константно време, а останалите (една много малка част) ще се изпълнят за линейно време.

Търсенето в `List<T>` е бавна операция, защото трябва да се обходят всички елементи. Изтриването по индекс или по стойност се изпълнява за линейно време. Изтриването е бавна операция, защото е свързана с преместване на всички елементи, които са след изтрития с една позиция наляво. Достъпът по индекс в `List<T>` става непосредствено, за константно време, тъй като елементите се съхраняват вътрешно в масив.

На практика `List<T>` комбинира добрите страни на масивите и на списъците, заради което е предпочитана структура данни в много ситуации. Например, ако трябва да обработим текстов файл и да извлечем от него всички думи, отговарящи на даден регулярен израз, най-удобната структура, в която можем да ги натрупваме, е `List<T>`, тъй като ни трябва списък, чиято дължина не е предварително известна и който да нараства динамично.

Динамичният масив (`List<T>`) е подходящ, когато трябва често да добавяме елементи и искаме да запазваме реда им на добавяне и да ги достъпваме често по индекс. Ако често търсим или изтриваме елемент, `List<T>` не е подходяща структура.

Ползвайте `List<T>`, когато трябва бързо да добавяте елементи и да ги достъпвате по индекс.

Стек (Stack)

Стекът е структура от данни, в която са дефинирани 3 операции: добавяне на елемент на върха на стека, изтриване на елемент от върха на стека и извличане на елемент от върха на стека без премахването му. Всички тези операции се изпълняват бързо, с константна сложност. Операциите търсене и достъп по индекс не се поддържат.

Стекът е структура с поведение LIFO (last in, first out) – последен влязъл, пръв излязъл. Използва се, когато трябва да моделираме такова поведение, например, ако трябва да пазим пътя до текущата позиция при рекурсивно търсене.

Ползвайте стек, когато е необходимо да реализирате поведението "последен влязъл, пръв излязъл" (LIFO).

Опашка (Queue)

Опашката е структура от данни, в която са дефинирани две операции: добавяне на елемент и извличане на елемента, който е наред. Тези две операции се изпълняват бързо, с константна сложност, тъй като опашката обикновено се имплементира чрез свързан списък. Припомняме, че свързаният списък може да добавя и изтрива бързо елементи в двата си края.

Поведението на структурата опашка е FIFO (first in, first out) – пръв влязъл, пръв излязъл. Операциите търсене и достъп по индекс не се поддържат. Опашката по естествен начин моделира списък от чакащи хора, задачи или други обекти, които трябва да бъдат обработени последователно, в реда на постъпването им.

Като пример за използване на опашка можем да посочим реализацията на алгоритъма "търсене в ширина", при който се започва от даден начален елемент и неговите съседи се добавят в опашка, след което се обработват по реда им на постъпване, а по време на обработката им техните съседи се добавят към опашката. Това се повтаря докато не се достигне до даден елемент, който търсим.

Ползвайте опашка, когато е необходимо да реализирате поведението "пръв влязъл, пръв излязъл" (FIFO).

Речник, реализиран с хеш-таблица (Dictionary<K,T>)

Структурата "речник" предполага съхраняване на двойки ключ-стойност и осигурява бързо търсене по ключ. При реализацията с хеш-таблица (класа `Dictionary<K,T>`) в .NET Framework) добавянето, търсенето и изтриването на елементи работят много бързо – със константна сложност в средния случай. Операцията достъп по индекс не е достъпна, защото елементите в хеш-таблицата се нареждат по почти случаен начин и редът им на постъпване не се запазва.

`Dictionary<K,T>` съхранява вътрешно елементите си в масив, като поставя всеки елемент на позиция, която се дава от хеш-функцията. По този начин масивът се запълва частично – в някои клетки има стойност, докато други стоят празни. Ако трябва да се поставят няколко стойности в една и съща клетка, те се нареждат в свързан списък (chaining). Това е един от начините за решаване на проблема с колизиите. Когато степента на запълненост на хеш-таблицата надвиши 100% (това е стойността по подразбиране на параметъра load factor), размерът ѝ нараства двойно и всички елементи заемат нови позиции. Тази операция работи с линейна сложност, но се изпълнява толкова рядко, че амортизираната сложност на операцията добавяне си остава константа.

Хеш-таблицата има една особеност: при неблагоприятно избрана хеш-функция, предизвикваща много колизии, основните операции могат да станат доста неефективни и да достигнат линейна сложност. В практиката, обаче, това почти не се случва. Затова се счита, че хеш-таблицата е най-бързата структура от данни, която осигурява добавяне и търсене по ключ.

Хеш-таблицата в .NET Framework предполага, че всеки ключ се среща в нея най-много веднъж. Ако запишем последователно два елемента с един и същ ключ, последният постъпил ще измести предходния и в крайна сметка ще изгубим единия елемент. Това е важна особеност, с която трябва да се съобразяваме.

Понякога се налага в един ключ да съхраняваме няколко стойности. Това не се поддържа стандартно, но можем да ползваме `List<T>` като стойност за този ключ и в него да натрупваме поредица от елементи. Например ако ни трябва хеш-таблица `Dictionary<int, string>`, в която да натрупваме двойки {цяло число, символен низ} с повторения, можем да ползваме `Dictionary<int, List<string>>`.

Хеш-таблица се препоръчва да се използва винаги, когато ни трябва бързо търсене по ключ. Например, ако трябва да преброим колко пъти се среща в текстов файл всяка дума измежду дадено множество думи, можем да ползваме `Dictionary<string, int>` като ползваме за ключ търсените думи, а за стойност – колко пъти се срещат във файла.

Ползвайте хеш-таблица, когато искате бързо да добавяте елементи и да търсите по ключ.

Много програмисти (най-вече начинаещите) живеят със заблудата, че основното предимство на хеш-таблицата е в удобството да търсим дадена стойност по нейния ключ. Всъщност основното предимство въобще не е това. Търсене по ключ можем да реализираме и с масив и със списък и дори със стек. Няма проблем, всеки може да ги реализира. Можем да си дефинираме клас `Entry`, който съхранява ключ и стойност и да си работим с масив или списък от `Entry` елементи. Можете да си реализираме търсене, но при всички положения то ще работи бавно. Това е големият проблем при списъците и масивите – не предлагат бързо търсене. За разлика от тях хеш-таблицата може да търси бързо и да добавя бързо нови елементи.

Основното предимство на хеш-таблицата пред останалите структури от данни е изключително бързото търсене и добавяне на елементи. Удобството на работа е второстепенен фактор.

Речник, реализиран с дърво (SortedDictionary<K,T>)

Реализацията на структурата от данни "речник" чрез червено-черно дърво (класът `SortedDictionary<K,T>`) е структура, която позволява съхранение на двойки ключ-стойност, при което ключовете са подредени (сортирани) по големина. Структурата осигурява бързо изпълнение на основните операции (добавяне на елемент, търсене по ключ и изтриване на елемент). Сложността, с която се изпълняват тези операции, е логаритмична – O(log(N)). Това означава 10 стъпки при 1000 елемента и 20 стъпки при 1 000 000 елемента.

За разлика от хеш-таблиците, където при лоша хеш-функция може да се достигне до линейна сложност на търсенето и добавянето, при структурата `SortedDictionary<K,T>` броят стъпки за изпълнение на основните операции в средния и в най-лошия случай е един и същ – $\log_2(N)$. При балансираните дървета няма хеширане, няма колизии и няма риск от използване на лоша хеш-функция.

Отново, както при хеш-таблиците, един ключ може да се среща в структурата най-много веднъж. Ако искаме да поставяме няколко стойности под един и същ ключ, трябва да ползваме за стойност на елементите някакъв списък, например `List<T>`.

`SortedDictionary<K,T>` държи вътрешно елементите си в червено-черно балансирано дърво, подредени по ключа. Това означава, че ако обходим структурата (чрез нейния итератор или чрез `foreach` цикъл в C#), ще получим елементите сортирани в нарастващ ред по ключа им. Понякога това може да е много полезно.

Използвайте `SortedDictionary<K,T>` в случаите, в които е необходима структура, в която бързо да добавяте, бързо да търсите и имате нужда от извличане на елементите, сортирани в нарастващ ред. В общия случай `Dictionary<K,T>` работи малко по-бързо от `SortedDictionary<K,T>` и е за предпочитане.

Като пример за използване на `SortedDictionary<K,T>` можем да дадем следната задача: да се намерят всички думи в текстов файл, които се срещат точно 10 пъти и да се отпечатат по азбучен ред. Това е задача, която можем да решим също така успешно и с `Dictionary<K,T>`, но ще ни се наложи да направим едно сортиране повече. При решението на тази задача можем да използваме `SortedDictionary<string, int>` и да преминем през всички думи от текстовия файл като за всяка от тях да запазваме в сортирания речник по колко пъти се среща във файла. След това можем да преминем през всички елементи на речника и да отпечатаме тези от тях, в които броят срещания е точно 10. Те ще бъдат подредени по азбучен ред, тъй като това в естествената вътрешна наредба на сортирания речник.

Използвайте `SortedDictionary<K,T>`, когато искате бързо да добавяте елементи и да търсите по ключ и елементите ще ви трябват след това сортирани по ключ.

Множество, реализирано с хеш-таблица (HashSet<T>)

Структурата от данни "множество" представлява съвкупност от елементи, сред които няма повтарящи се. Основните операции са добавяне на елемент към множеството, проверка за принадлежност на елемент към множеството (търсене) и премахване на елемент от множеството (изтриване). Операцията търсене по индекс не се поддържа, т.е. нямаме директен достъп до елементите по пореден номер, защото в тази структура поредни номера няма.

Множество, реализирано чрез хеш-таблица (класът **`HashSet<T>`**), е частен случай на хеш-таблица, при който имаме само ключове, а стойностите, записани под всеки ключ са без значение. Този клас е включен в .NET Framework едва от версия 3.5 нататък.

Както и при хеш-таблицата, основните операции в структурата от данни **`HashSet<T>`** са реализирани с константна сложност O(1). Както и при хеш-таблицата, при неблагоприятна хеш-функция може да се стигне до линейна сложност на основните операции, но в практиката това почти не се случва.

Като пример за използването на **`HashSet<T>`** можем да посочим задачата за намиране на всички различни думи в даден текстов файл.

Ползвайте `HashSet<T>`, когато трябва бързо да добавяте елементи към множество и да проверявате дали даден елемент е от множеството.

Множество, реализирано с дърво (SortedSet<T>)

Множество, реализирано чрез червено-черно дърво, е частен случай на **`SortedDictionary<K,T>`**, в който ключовете и стойностите съвпадат.

Както и при **`SortedDictionary<K,T>`** структурата, основните операции в **`SortedSet<T>`** са реализирани с логаритмична сложност O(log(N)), като тази сложност е една и съща и в средния и в най-лошия случай.

Като пример за използването на **`SortedSet<T>`** можем да посочим задачата за намиране на всички различни думи в даден текстов файл и отпечатването им подредени по азбучен ред.

Използвайте `SortedSet<T>`, когато трябва бързо да добавяте елементи към множество и да проверявате дали даден елемент е от множеството и освен това елементите ще ви трябват сортирани в нарастващ ред.

Избор на структура от данни – примери

Сега ще дадем няколко задачи, при които изборът на подходяща структура от данни е от решаващо значение за ефективността на тяхното решение. Целта е да ви покажем типични ситуации, в които се използват разгледаните структури от данни и да ви научим в какви ситуации какви структури от данни да използвате.

Генериране на подмножества

Дадено е множество от символни низове S, например S = {море, бира, пари, кеф}. Да се напише програма, която отпечатва всички подмножества на S.

Задачата има много и различни по идея решения, но ние ще се спрем на следното решение: Започваме от празно подмножество (с 0 елемента):

```
{ }
```

Към него добавяме всеки от елементите на S и получаваме съвкупност от подмножества с по 1 елемент:

{море}, {бира}, {пари}, {кеф}

Към всяко от получените едноелементни подмножества добавяме всеки от елементите на S, който все още не се съдържа в съответното подмножество и получаваме всички двуелементни подмножества:

{море, бира}, {море, пари}, {море, кеф}, {бира, пари}, {бира, кеф}, {пари, кеф}

Ако продължим по същия начин, ще получим всички 3-елементни подмножества и след тях 4-елементните т. н. до N-елементните подмножества.

Как да реализираме този алгоритъм? Трябва да изберем подходящи структури от данни, нали?

Можем да започнем с избора на структурата, която съхранява началното множество от елементи S. Тя може да е масив, свързан списък, динамичен масив (**`List<string>`**) или множество, реализирано като **`SortedSet<string>`** или **`HashSet<string>`**. За да си отговорим на въпроса коя структура е най-подходяща, нека помислим кои са операциите, които ще трябва да извършваме върху тази структура. Сещаме се само за една операция – обхождане на всички елементи на S. Тази операция може да бъде реализирана ефективно с всяка от изброените структури. Избираме масив, защото е най-простата структура от възможните и с него се работи най-лесно.

Следва да изберем структурата, в която ще съхраняваме едно от подмножествата, които генерираме, например {море, кеф}. Отново си задаваме въпроса какви са операциите, които извършваме върху такова подмножество от думи. Операциите са проверка за съществуване на елемент и добавяне на елемент, нали? Коя структура реализира бързо тази двойка операции? Масивите и списъците не търсят бързо, речниците съхраняват двойки ключ-стойност, което не е нашия случай. Остана да видим структурата множество. Тя поддържа бързо търсене и бързо добавяне. Коя имплементация да изберем – **`SortedSet<string>`** или **`HashSet<string>`**? Нямаме изискване за сортиране на думите по азбучен ред, така че избираме по-бързата имплементация – **`HashSet<string>`**.

Остана да изберем още една структура от данни – структурата, в която съхраняваме съвкупност от подмножества от думи, например:

> {море, бира}, {море, пари}, {море, кеф}, {бира, пари}, {бира, кеф}, {пари, кеф}

В тази структура трябва да можем да добавяме, както и да обхождаме елементите й последователно. На тези изисквания отговарят структурите списък, стек, опашка и множество. Във всяка от тях можем да добавяме бързо и да обхождаме елементите й. Ако разгледаме внимателно алгоритъма за генериране на подмножествата, ще забележим, че всяко от тях се обработва в стил "пръв генериран, пръв обработен". Подмножеството, което първо е било получено първо, се обработва първо и от него се получават подмножествата с 1 елемент повече, нали? Следователно на нашия алгоритъм най-точно ще пасне структурата от данни опашка. Можем да опишем алгоритъма така:

1. Започваме от опашка, съдържаща празното множество {}.
2. Взимаме поредния елемент **`subset`** от опашката и към него се опитваме да добавим всеки елемент от S, който не се съдържа в **`subset`**. Резултатът е множество, което добавяме към опашката.
3. Повтаряме предходната стъпка докато опашката свърши.

Виждате, че с разсъждения стигнахме до класическия алгоритъм "търсене в ширина". След като знаем какви структури от данни да използваме, имплементацията става бързо и лесно. Ето как би могла да изглежда тя:

```
string[] words = {"море", "бира", "пари", "кеф"};
Queue<HashSet<string>> subsetsQueue =
   new Queue<HashSet<string>>();
HashSet<string> emptySet = new HashSet<string>();
subsetsQueue.Enqueue(emptySet);
while (subsetsQueue.Count > 0)
```

```
{
   HashSet<String> subset = subsetsQueue.Dequeue();

   // Print current subset
   Console.Write("{ ");
   foreach (string word in subset)
   {
      Console.Write("{0} ", word);
   }
   Console.WriteLine("}");

   // Generate and enqueue all possible child subsets
   foreach (string element in words)
   {
      if (! subset.Contains(element))
      {
         HashSet<string> newSubset = new HashSet<string>();
         newSubset.UnionWith(subset);
         newSubset.Add(element);
         subsetsQueue.Enqueue(newSubset);
      }
   }
}
```

Ако изпълним горния код, ще се убедим, че той генерира успешно всички подмножества на S, но някои от тях ги генерира по няколко пъти:

```
{ }
{ море }
{ бира }
{ пари }
{ кеф }
{ море бира }
{ море пари }
{ море кеф }
{ бира море }
...
```

В примера множествата `{ море бира }` и `{ бира море }` са всъщност едно и също множество. Изглежда не сме се сетили за повторенията, които се получават при разбъркване на реда на елементите на едно и също множество. Как можем да ги избегнем?

Да номерираме думите по техните индекси:

море → 0
бира → 1
пари → 2
кеф → 3

Понеже подмножествата {1, 2, 3} и {2, 1, 3} са всъщност едно и също подмножество, за да нямаме повторения, ще наложим изискването да генерираме само подмножества, в които индексите са подредени по големина. Можем вместо множества от думи да пазим множества от индекси, нали? В тези множества от индекси ни трябват две операции: добавяне на индекс и взимане на най-големия индекс, за да добавяме само индекси, по-големи от него. Очевидно **`HashSet<T>`** вече не ни върши работа, но можем успешно да ползваме **`List<T>`**, в който елементите са наредени по големина и най-големият елемент по естествен начин е последен в списъка.

В крайна сметка нашия алгоритъм добива следната форма:

1. Нека N е броят елементи в S. Започваме от опашка, съдържаща празния списък {}.
2. Взимаме поредния елемент **`subset`** от опашката. Нека **`start`** е най-големия индекс в **`subset`**. Към **`subset`** добавяме всички индекси, които са по-големи от **`start`** и по-малки от N. В резултат получаваме няколко нови подмножества, които добавяме към опашката.
3. Повтаряме последната стъпка докато опашката свърши.

Ето как изглежда реализацията на новия алгоритъм:

```
using System;
using System.Collections.Generic;

public class Subsets
{
  static string[] words = { "море", "бира", "пари", "кеф" };

  static void Main()
  {
    Queue<List<int>> subsetsQueue = new Queue<List<int>>();
    List<int> emptySet = new List<int>();
    subsetsQueue.Enqueue(emptySet);
    while (subsetsQueue.Count > 0)
    {
      List<int> subset = subsetsQueue.Dequeue();
      Print(subset);
      int start = -1;
      if (subset.Count > 0)
      {
        start = subset[subset.Count - 1];
      }
      for (int i = start + 1; i < words.Length; i++)
      {
        List<int> newSubset = new List<int>();
        newSubset.AddRange(subset);
        newSubset.Add(i);
```

```
            subsetsQueue.Enqueue(newSubset);
         }
      }
   }

   static void Print(List<int> subset) {
      Console.Write("[ ");
      for (int i=0; i<subset.Count; i++) {
         int index = subset[i];
         Console.Write("{0} ", words[index]);
      }
      Console.WriteLine("]");
   }
}
```

Ако изпълним програмата, ще получим очаквания коректен резултат:

```
[ ]
[ море ]
[ бира ]
[ пари ]
[ кеф ]
[ море бира ]
[ море пари ]
[ море кеф ]
[ бира пари ]
[ бира кеф ]
[ пари кеф ]
[ море бира пари ]
[ море бира кеф ]
[ море пари кеф ]
[ бира пари кеф ]
[ море бира пари кеф ]
```

Подреждане на студенти

Даден е текстов файл, съдържащ данните за група студенти и курсовете, които те изучават, разделени с |. Файлът изглежда по следния начин:

```
Кирил  | Иванов  | C#
Милена | Стефанова | PHP
Кирил | Иванов| Java
Петър  | Иванов  | C#
Стефка | Василева  | Java
Милена | Василева  | C#
Милена | Стефанова | C#
```

Да се напише програма, която отпечатва всички курсове и за всеки от тях студентите, които са ги записали, подредени първо по фамилия, след това по име (ако фамилиите съвпадат).

Задачата можем да реализираме чрез хеш-таблица, която по име на курс пази списък от студенти. Избираме хеш-таблица, защото в нея можем бързо да търсим по име на курс.

За да изпълним условието за подредба по фамилия и име, при отпечатването на студентите от всеки курс ще трябва да сортираме съответния списък. Другият вариант е да ползваме **`SortedSet<T>`** за студентите от всеки курс (понеже той вътрешно е сортиран), но понеже може да има студенти с еднакви имена, трябва да ползваме **`SortedSet<List<String>>`**. Става твърде сложно. Избираме по-лесния вариант – да ползваме **`List<Student>`** и да го сортираме преди да го отпечатаме.

При всички случаи ще трябва да реализираме интерфейса **`IComparable`**, за да дефинираме наредбата на елементите от тип **`Student`** според условието на задачата. Необходимо е първо да сравняваме фамилията и при еднаква фамилия да сравняваме след това името. Напомняме, че за да сортираме елементите на даден клас в нарастващ ред, е необходимо изрично да дефинираме логиката на тяхната наредба. В .NET Framework това става чрез интерфейса **`IComparable<T>`**. Нека дефинираме класа **`Student`** и имплементираме **`IComparable<Student>`**. Получаваме нещо такова:

```
public class Student : IComparable<Student>
{
  private string firstName;
  private string lastName;

  public Student(string firstName, string lastName)
  {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public int CompareTo(Student student)
  {
    int result = lastName.CompareTo(student.lastName);
    if (result == 0)
    {
      result = firstName.CompareTo(student.firstName);
    }
    return result;
  }

  public override String ToString()
  {
    return firstName + " " + lastName;
  }
}
```

Сега вече можем да напишем кода, който прочита студентите и техните курсове и ги записва в хеш-таблица, която по име на курс пази списък със студентите в този курс (**Dictionary<string, ArrayList<Student>>**). След това вече е лесно – итерираме по курсовете, сортираме студентите и ги отпечатваме:

```
// Read the file and build the hash-table of courses
Dictionary<string, List<Student>> courses =
  new Dictionary<string, List<Student>>();
StreamReader reader = new StreamReader("Students.txt",
  Encoding.GetEncoding("windows-1251"));
using (reader)
{
  while (true)
  {
    string line = reader.ReadLine();
    if (line == null)
    {
      break;
    }
    string[] entry = line.Split(new char[] { '|' });
    string firstName = entry[0].Trim();
    string lastName = entry[1].Trim();
    string course = entry[2].Trim();
    List<Student> students;
    if (! courses.TryGetValue(course, out students))
    {
      // New course -> create a list of students for it
      students = new List<Student>();
      courses.Add(course, students);
    }
    Student student = new Student(firstName, lastName);
    students.Add(student);
  }
}

// Print the courses and their students
foreach (string course in courses.Keys)
{
  Console.WriteLine("Course " + course + ":");
  List<Student> students = courses[course];
  students.Sort();
  foreach (Student student in students)
  {
    Console.WriteLine("\t{0}", student);
  }
}
```

Примерният код чете студентите от файла **Students.txt** като изрично задава кодиране "windows-1251" за да се прочете правилно кирилицата. След това парсва редовете му последователно един по един като ги разделя по вертикална черта "|" и след това ги изчиства от интервали в началото и в края. След прочитането на всеки студент се проверява хеш-таблицата дали съдържа неговия курс. Ако курсът е намерен, студентът се добавя към списъка със студенти за този курс. Ако курсът не е намерен, се създава нов списък, към него се добавя студента и списъкът се записва в хеш-таблицата под ключ името на курса.

Отпечатването на курсовете и студентите не е сложно. От хеш-таблицата се извличат всички ключове. Това са имената на курсовете. За всеки курс се извлича списък от студентите му, те се сортират и се отпечатват. Сортирането става с вградения метод **`Sort()`**, като се използва метода за сравнение **`CompareTo(…)`** от интерфейса **`IComparable<T>`** както е дефинирано в класа **`Student`** (сравнение първо по фамилия, а при еднакви фамилии – по име). Накрая сортираните студенти се отпечатват чрез предефинирания в тях виртуален метод **`ToString()`**. Ето как изглежда изходът от горната програма:

```
Course C#:
        Милена Василева
        Кирил Иванов
        Петър Иванов
        Милена Стефанова
Course PHP:
        Милена Стефанова
Course Java:
        Стефка Василева
        Кирил Иванов
```

Подреждане на телефонен указател

Даден е текстов файл, който съдържа имена на хора, техните градове и телефони. Файлът изглежда по следния начин:

```
Киро | Варна    | 052 / 23 45 67
Пешо | София    | 02 / 234 56 78
Мими | Пловдив  | 0888 / 22 33 44
Лили | София    | 0899 / 11 22 33
Дани | Варна    | 0897 / 44 55 66
```

Да се напише програма, която отпечатва всички градове по азбучен ред и за всеки от тях отпечатва всички имена на хора по азбучен ред и съответния им телефон.

Задачата можем да решим по много начини, например като сортираме по два критерия: на първо място по град и на второ място по телефон и след това отпечатаме телефонния указател.

Нека, обаче решим задачата без сортиране, като използваме стандартните структури от данни в .NET Framework. Искаме да имаме в сортиран вид градовете. Това означава, че е най-добре да ползваме структура, която държи вътрешно елементите си в сортиран вид. Такава е например балансираното дърво – **`SortedSet<T>`** или **`SortedDictionary<K,T>`**. Понеже всеки запис от телефонния указател съдържа освен град и други данни, е по-удобно да имаме **SortedDictionary<K,T>**, който по ключ име на град пази списък от хора и техните телефони. Понеже искаме списъкът на хората за всеки град също да е сортиран по азбучен ред по имената на хората, можем отново да ползваме структурата **SortedDictionary<K,T>**. Като ключ можем да слагаме име на човек, а като стойност – неговия телефон. В крайна сметка получаваме структурата **SortedDictionary<string, SortedDictionary<string, string>>**. Следва примерна имплементация, която показва как можем да решим задачата с тази структура:

```
// Read the file and build the phone book
SortedDictionary<string, SortedDictionary<string, string>>
  phonesByTown = new SortedDictionary<string,
    SortedDictionary<string, string>>();
StreamReader reader = new StreamReader("PhoneBook.txt",
  Encoding.GetEncoding("windows-1251"));
using (reader)
{
  while (true)
  {
    string line = reader.ReadLine();
    if (line == null)
    {
      break;
    }
    string[] entry = line.Split(new char[]{'|'});
    string name = entry[0].Trim();
    string town = entry[1].Trim();
    string phone = entry[2].Trim();

    SortedDictionary<string, string> phoneBook;
    if (! phonesByTown.TryGetValue(town, out phoneBook))
    {
      // This town is new. Create a phone book for it
      phoneBook = new SortedDictionary<string, string>();
      phonesByTown.Add(town, phoneBook);
    }
    phoneBook.Add(name, phone);
  }
}

// Print the phone book by towns
foreach (string town in phonesByTown.Keys)
{
  Console.WriteLine("Town " + town + ":");
  SortedDictionary<string, string> phoneBook =
    phonesByTown[town];
  foreach (var entry in phoneBook)
  {
        string name = entry.Key;
        string phone = entry.Value;
        Console.WriteLine("\t{0} - {1}", name, phone);
    }
}
```

Ако изпълним този примерен код с вход примерния телефонен указател, ще получим очаквания резултат:

```
Town Варна:
    Дани - 0897 / 44 55 66
    Киро - 052 / 23 45 67
Town Пловдив:
    Мими - 0888 / 22 33 44
Town София:
    Лили - 0899 / 11 22 33
    Пешо - 02 / 234 56 78
```

Търсене в телефонен указател

Ще даден още един пример, за да затвърдим начина, по който разсъждаваме, за да изберем подходящи структури от данни. Даден е телефонен указател, записан в текстов файл, който съдържа имена на хора, техните градове и телефони. Имената на хората могат да бъдат във формат малко име или прякор или име + фамилия или име + презиме + фамилия. Файлът би могъл да има следния вид:

```
Киро Киров        | Варна    | 052 / 23 45 67
Мундьо            | София    | 02 / 234 56 78
Киро Киров Иванов | Пловдив  | 0888 / 22 33 44
Лили Иванова      | София    | 0899 / 11 22 33
Киро              | Плевен   | 064 / 88 77 66
Киро бирата       | Варна    | 0897 / 44 55 66
Киро              | Плевен   | 0897 / 44 55 66
```

Възможно е да има няколко души, записани под едно и също име, дори и от един и същ град. Възможно е някой да има няколко телефона и в такъв случай той се изписва няколко пъти във входния файл. Телефонният указател може да бъде доста голям (до 1 000 000 записа).

Даден е файл със заявки за търсене. Заявките са два вида:

- Търсене по име / прякор / презиме / фамилия. Заявката има вида **`list(name)`**.
- Търсене по име / прякор / презиме / фамилия + град. Заявката има вида **`find(name, town)`**.

Ето примерен файл със заявки:

```
list(Киро)
find(Пешо, София)
list(Лили)
list(Киров)
find(Иванов, Пловдив)
list(Баба)
```

Да се напише програма, която по даден телефонен указател и файл със заявки да върне всички отговори на заявките за търсене. За всяка заявка да се изведе списък от записите в телефонния указател, които й съответстват или съобщението "Not found", ако заявката не намира нищо. Заявките могат да са голям брой (например 50 000).

Тази задача не е толкова лесна, колкото предходните. Едно лесно за реализация решение е при всяка заявка да се сканира целият телефонен указател и да се изваждат всички записи, в които има съвпадения с търсената информация. Това, обаче ще работи бавно, защото записите могат да са много и заявките също могат да са много. Необходимо е да намерим начин да търсим бързо, без да сканираме всеки път целия телефонен указател.

В хартиените телефонни указатели телефоните са дадени по имената на хората, подредени в азбучен ред. Сортирането няма да ни помогне, защото някой може да търси по име, друг по фамилия, а трети – по прякор и име на град. Ние трябва да можем да търсим по всичко това едновременно. Въпросът е как да го направим?

Ако поразсъждаваме малко, ще се убедим, че в задачата се изисква търсене по всяка от думите, които се срещат в първата колона на телефонния указател и евентуално по комбинацията дума от първата колона и град от втората колона. Знаем, че най-бързото търсене, което познаваме, се реализира с хеш-таблица. Добре, но какво да използваме за ключ и какво да използваме за стойност в хеш-таблицата?

Дали пък да не ползваме няколко хеш-таблици: една за търсене по първата дума от първата колона, още една за търсене по втората колона, една за търсене по град и т.н. Ако се замислим още малко, ще си зададем въпроса – защо са ни няколко хеш-таблици? Не може ли да търсим само в една хеш-таблица. Ако имаме "Петър Иванов", в таблицата ще сложим под ключ "Петър" неговия телефон и същевременно под ключ "Иванов" същия телефон. Ако някой търси една от двете думи, ще намери телефона на Петър.

До тук добре, обаче как ще търсим по име и по град, например "Петър от Варна"? Възможно е първо да намерим всички с име "Петър" и от тях да отпечатаме само тези, които са от Варна. Това ще работи, но ако има достатъчно много хора с име Петър, търсенето по град ще работи бавно. Тогава защо не направим хеш-таблица по ключ име на човек и стойност друга хеш-таблица, която по град връща списък от телефони? Това би трябвало да работи. Нещо подобно правихме в предходната задача, нали?

Може ли да ни хрумне нещо още по-умно? Не може ли в основната хеш-таблица за телефонния указател да сложим под ключ "Петър от Варна" телефоните на всички, които се казват Петър и са от Варна? Изглежда това ще реши проблема и ще можем да използваме само една хеш-таблица за всички търсения.

Използвайки последната идея стигаме до следния алгоритъм: четем ред по ред телефонния указател и за всяка дума от името на човека **d_1**, **d_2**, **…**, **d_k** и за всеки град **t** добавяме текущия запис от указателя под следните ключове: **d_1**, **d_2**, **…**, **d_k**, "**d_1 от t**", "**d_2 от t**", …, "**d_k от t**". Така си гарантираме, че ще можем да търсим по всяко от имената на съответния човек и по всяка двойка име + град. За да можем да търсим без значение на регистъра (главни или малки букви), можем да направим предварително всички букви малки. След това търсенето е тривиално – просто търсим в хеш-таблицата подадената дума **d** или ако ни подадат дума **d** и град **t**, търсим по ключ "**d от t**". Понеже за един и същ ключ може да има много телефони, ползваме за стойност в хеш-таблицата списък от символни низове (**List<string>**). Нека разгледаме една имплементация на описания алгоритъм:

```
class PhoneBookFinder
{
  const string PhoneBookFileName = "PhoneBook.txt";
  const string QueriesFileName = "Queries.txt";

  static Dictionary<string, List<string>> phoneBook =
    new Dictionary<string, List<string>>();

  static void Main()
  {
    ReadPhoneBook();
    ProcessQueries();
  }

  static void ReadPhoneBook()
  {
    StreamReader reader = new StreamReader(
      PhoneBookFileName, Encoding.GetEncoding("windows-1251"));
    using (reader)
    {
      while (true)
      {
        string line = reader.ReadLine();
        if (line == null)
        {
          break;
        }
        string[] entry = line.Split(new char[]{'|'});
        string names = entry[0].Trim();
        string town = entry[1].Trim();

        string[] nameTokens =
          names.Split(new char[] {' ', '\t'} );
        foreach (string name in nameTokens)
        {
          AddToPhoneBook(name, line);
          string nameAndTown = CombineNameAndTown(town, name);
          AddToPhoneBook(nameAndTown, line);
        }
      }
    }
  }

  static string CombineNameAndTown(string town, string name)
  {
    return name + " от " + town;
  }

  static void AddToPhoneBook(string name, string entry)
  {
    name = name.ToLower();
    List<string> entries;
    if (! phoneBook.TryGetValue(name, out entries))
    {
      entries = new List<string>();
      phoneBook.Add(name, entries);
    }
    entries.Add(entry);
  }

  static void ProcessQueries()
  {
    StreamReader reader = new StreamReader(QueriesFileName,
      Encoding.GetEncoding("windows-1251"));
    using (reader)
    {
      while (true)
      {
        string query = reader.ReadLine();
        if (query == null)
        {
          break;
        }
        ProcessQuery(query);
      }
    }
  }

  static void ProcessQuery(string query)
```

```
    {
      if (query.StartsWith("list("))
      {
        int listLen = "list(".Length;
        string name = query.Substring(
          listLen, query.Length-listLen-1);
        name = name.Trim().ToLower();
        PrintAllMatches(name);
      }
      else if (query.StartsWith("find("))
      {
        string[] queryParams = query.Split(
          new char[] { '(', ' ', ',', ')' },
          StringSplitOptions.RemoveEmptyEntries);
        string name = queryParams[1];
        name = name.Trim().ToLower();
        string town = queryParams[2];
        town = town.Trim().ToLower();
        string nameAndTown =
          CombineNameAndTown(town, name);
        PrintAllMatches(nameAndTown);
      }
      else
      {
        Console.WriteLine(
          query + " is invalid command!");
      }
    }

    static void PrintAllMatches(string key)
    {
      List<string> allMatches;
      if (phoneBook.TryGetValue(key, out allMatches))
      {
        foreach (string entry in allMatches)
        {
          Console.WriteLine(entry);
        }
      }
      else
      {
        Console.WriteLine("Not found!");
      }
      Console.WriteLine();
    }
  }
```

При прочитането на телефонния указател чрез разделяне по вертикална черта "|" от него се извличат трите колони (име, град и телефон) от всеки негов ред. След това името се разделя на думи и всяка от думите се добавя в хеш-таблицата. Допълнително се добавя и всяка дума, комбинирана с града (за да можем да търсим по двойката име + град).

Следва втората част на алгоритъма – изпълнението на командите. В нея файлът с командите се чете ред по ред и всяка команда се обработва. Обработката включва парсване на командата, извличането на име или име и град от нея и търсене по даденото име или име, комбинирано с града. Търсенето се извършва директно в хеш-таблицата, която се създава при прочитане на телефонния указател.

За да се игнорират разликите между малки и главни букви, всички ключове в хеш-таблицата се добавят с малки букви и при търсенето ключовете се търсят също с малки букви.

Избор на структури от данни – изводи

От множеството примери става ясно, че изборът на подходяща структура от данни силно зависи от конкретната задача. Понякога се налага структурите от данни да се комбинират или да се използват едновременно няколко структури.

Кога каква структура да подберем зависи най-вече от операциите, които извършваме, така че винаги си задавайте въпроса "какви операции трябва да изпълнява ефективно структурата, която ми трябва". Ако знаете операциите, лесно може да съобразите коя структура ги изпълнява най-ефективно и същевременно е лесна и удобна за ползване.

За да изберете ефективно структура от данни, трябва първо да измислите алгоритъма, който ще имплементирате и след това да потърсите подходящите структури за него.

Тръгвайте винаги от алгоритъма към структурите от данни, а не обратното.

Външни библиотеки с .NET колекции

Добре известен факт е, че библиотеката със стандартни структури от данни в .NET Framework `System.Collections.Generic` е доста бедна откъм функционалност. В нея липсват имплементации на основни концепции в структурите данни, мултимножества и приоритетни опашки, за които би трябвало да има както стандартни класове, така и базови системни интерфейси.

Когато ни се наложи да използваме структура от данни, която стандартно не е имплементирана в .NET Framework имаме два варианта:

- Първи вариант: имплементираме си сами структурата от данни. Това дава гъвкавост, тъй като имплементацията ще е съобразена напълно с нашите нужди, но отнема много време и има голяма вероятност от допускане на грешки. Например, ако трябва да се имплементира по кадърен начин балансирано дърво, това може да отнеме на добър програмист няколко дни (заедно с тестовете). Ако се имплементира от неопитен програмист ще отнеме още повече време и има огромна вероятност в имплементацията да има грешки.

- Вторият вариант (като цяло за предпочитане): да си намерим външна библиотека, в която е реализирана на готово нужната ни функционалност. Този подход има предимството, че ни спестява време и проблеми, тъй като готовите библиотеки със структури от данни в повечето случаи са добре тествани. Те са използвани години наред от хиляди разработчици и това ги прави зрели и надеждни.

Power Collections for .NET

Една от най-популярните и най-пълни библиотеки с ефективни реализации на фундаментални структури от данни за C# и .NET разработчици е проектът с отворен код "Wintellect's Power Collections for .NET" – http://powercollections.codeplex.com/. Той предоставя свободна, надеждна, ефективна, бърза и удобна имплементация на следните често използвани структури от данни, които липсват или са непълно имплементирани в .NET Framework:

- **`Set<T>`** – множество от елементи, имплементирано чрез хеш-таблица. Реализира по ефективен начин основните операции над множества: добавяне на елемент, изтриване на елемент, търсене на елемент, обединение, сечение и разлика на множества и други. По функционалност и начин на работа класът прилича на стандартния клас **`HashSet<T>`** в .NET Framework.

- **`Bag<T>`** – мултимножество от елементи (множество с повторения), имплементирано чрез хеш-таблица. Реализира ефективно всички основни операции с мултимножества.

- **`OrderedSet<T>`** – подредено множество от елементи (без повторения), имплементирано чрез балансирано дърво. Реализира ефективно всички основни операции с множества и при обхождане връща елементите си в нарастващ ред (според използвания компаратор). Позволява бързо извличане на подмножества от стойностите в даден интервал от стойности.

- **`OrderedBag<T>`** – подредено мултимножество от елементи, имплементирано чрез балансирано дърво. Реализира ефективно всички основни операции с мултимножества и при обхождане връща елементите си в нарастващ ред (според използвания компаратор). Позволява бързо извличане на подмножества от стойностите в даден интервал от стойности.

- **`MultiDictionary<K,T>`** – представлява хеш-таблица, позволяваща повторения на ключовете. За един ключ се пази съвкупност от стойности, а не една единична стойност.

- **`OrderedDictionary<K,T>`** – представлява речник, реализиран с балансирано дърво. Позволява бързо търсене по ключ и при обхождане на елементите ги връща сортирани в нарастващ ред. Позволява бързо извличане на елементите в даден диапазон от ключове. По функционалност и начин на работа класът прилича на стандартния клас **`SortedDictionary<K,T>`** в .NET Framework.

- **`Deque<T>`** – представлява ефективна реализация на опашка с два края (double ended queue), която на практика комбинира структурите стек и опашка. Позволява ефективно добавяне, извличане и изтриване на елементи в двата края.

- **`BagList<T>`** – списък от елементи, достъпни по индекс, който позволява бързо вмъкване и изтриване на елемент от определена позиция. Операциите достъп по индекс, добавяне, вмъкване на позиция и изтриване от позиция имат сложност O(log N). Реализацията е с балансирано дърво. Структурата е добра алтернатива на **`List<T>`**, при която вмъкването и изтриването от определена позиция отнема линейно време поради нуждата от преместване на линеен брой елементи наляво или надясно.

Оставяме на читателя възможността да си изтегли библиотеката "Power Collections for .NET" от нейния сайт и да експериментира с нея. Тя може да е много полезна при решаването на някои задачи от упражненията.

Упражнения

1. Хеш-таблиците не позволяват в един ключ да съхраняваме повече от една стойност. Как може да се заобиколи това ограничение?

2. Реализирайте структура от данни, която изпълнява бързо следните две операции: добавяне на елемент и извличане на най-малкия елемент. Структурата трябва да позволява включването на повтарящи се елементи.

3. Текстов файл **`students.txt`** съдържа информация за студенти и техните специалност в следния формат:

```
Spas Delev | Computer Sciences
Ivan Ivanov | Software Engeneering
Gergana Mineva | Public Relations
Nikolay Kostov | Computer Sciences
Stanimira Georgieva | Public Relations
Vasil Ivanov | Software Engeneering
```

   Като използвате **`SortedDictionary<K,T>`** изведете на конзолата в азбучен ред специалностите и за всеки от тях изведете имената на студентите, сортирани първо по фамилия, после по първо име, както е показано:

```
Computer Sciences: Spas Delev, Nikolay Kostov
Public Relations: Stanimira Georgieva, Gergana Mineva
Software Engeneering: Ivan Ivanov, Vasil Ivanov
```

4. Имплементирайте клас **`BiDictionary<K1,K2,T>`**, който позволява добавяне на тройки **`{key1, key2, value}`** и бързо търсене по ключовете **`key1`**, **`key2`** и търсене по двата ключа. Заб.: Разрешено е добавянето на много елементи с един и същ ключ.
5. В една голяма верига супермаркети се продават милиони стоки. Всяка от тях има уникален номер (баркод), производител, наименование и цена. Каква структура от данни можем да използваме, за да можем бързо да намерим всички стоки, които струват между 5 и 10 лева?
6. Голяма компания за продажби притежава милиони статии, всяка от които има баркод, производител, заглавие и цена. Имплементирайте структура от данни, която позволява бързи заявки за статии по цена на статията в определен интервал [x…y].
7. Разписанието на дадена конгресна зала представлява списък от събития във формат [начална дата и час; крайна дата и час; наименование на събитието]. Какви структури от данни можем да ползваме, за да можем бързо да проверим дали залата е свободна в даден интервал [начална дата и час; крайна дата и час]?
8. Имплементирайте структурата от данни **`PriorityQueue<T>`**, която предоставя бърз достъп за изпълнение на следните операции: добавяне на елемент, изкарване на най-малкия елемент.
9. Представете си, че разработвате търсачка в обявите за продажба на коли на старо, която обикаля десетина сайта за обяви и събира от тях всички обяви за последните няколко години. След това търсачката позволява бързо търсене по един или няколко критерии: марка, модел, цвят, година на производство и цена. Нямате право да ползвате система за управление на бази от данни и трябва да реализирате собствено индексиране на обявите в паметта, без да пишете на твърдия диск и без да използвате LINQ. При търсене по цена се подава минимална и максимална цена. При търсене по година на производство се задава начална и крайна година. Какви структури от данни ще ползвате, за да осигурите бързо търсене по един или няколко критерия?

Решения и упътвания

1. Можете да използвате **`Dictionary<key, List<value>>`** или да си създадете собствен клас **`MyCollection`**, който да се грижи за стойностите с еднакъв ключ и да използвате **`Dictionary<key, MyCollection>`**.

2. Можете да използвате **SortedSet<List<int>>** и неговите операции **Add()** и **First()**. Задачата има и по-ефективно решение – структурата от данни "двоична пирамида" (binary heap). Можете да прочетете за нея от Уикипедия: http://en.wikipedia.org/wiki/Binary_heap.

3. Задачата е много подобна на тази от секцията "Подреждане на студенти".

4. Едно от решенията на тази задача е да използвате две инстанции на класа **Dictionary** по една за всеки от двата ключа и когато добавяте или махате елемент от **BiDictionary**, съответно да го добавяте или махате и в двете хеш-таблици. Когато търсите по първия или по втория ключ, ще гледате за елементи съответно в първата или втората хеш-таблица, а когато търсите за елемент по двата ключа, ще гледате в двете хеш-таблици и ще връщате само елементите, които се намират и в двете намерени множества.

5. Ако държим стоките в сортиран по цена масив (например в структура **List<Product>**, който първо запълваме и накрая сортираме), за да намерим всички стоки, които струват между 5 и 10 лева можем два пъти да използваме двоично търсене. Първо можем да намерим най-малкия индекс **start**, на който стои стока струваща най-малко 5 лева. След това можем да намерим най-големия индекс **end**, на който стои стока струваща най-много 10 лева. Всички стоки на позиции в интервала [**start** … **end**] струват между 5 и 10 лв. За двоично търсене в сортиран масив можете да прочетете в Уикипедия: http://en.wikipedia.org/wiki/Binary_search.

   Като цяло подходът с използването на сортиран масив и двоично търсене в него работи отлично, но има един недостатък: в сортиран масив добавянето на нов елемент е много бавна операция, тъй като изисква преместване на линеен брой елементи с една позиция напред спрямо вмъкнатия нов елемент. Класовете **SortedDictionary<T,K>** и **SortedSet<T>** съответно позволяват бързо вмъкване, запазващо елементите в сортиран вид, но не позволява достъп по индекс, двоично търсене и съответно извличане на всички елементи, които са по-големи или по-малки от даден ключ. Това е ограничение на имплементацията на тези структури в .NET Framework. В класическите имплементации на балансирано дърво има стандартна операция за бързо извличане на поддърво с елементите в даден интервал от два ключа. Това е реализирано например в класа **OrderedSet<T>** от библиотеката "Wintellect's Power Collections for .NET" (http://www.codeplex.com/PowerCollections).

6. Използвайте структурата от данни **OrderedMultiDictionary** от Wintellect's Power Collections.

7. Можем да конструираме два сортирани масива (**List<Event>**): единият да пази събитията сортирани в нарастващ ред по ключ началната дата и час, а другият да пази същите събития сортирани по ключ крайна дата и час. Можем да намерим чрез двоично търсене всички събития,

които се съдържат частично или изцяло между два момента от времето [`start`, `end`] по следния начин:

- Намираме всички събития, завършващи след момента `start` (чрез двоично търсене).
- Намираме всички събития, започващи преди момента `end` (чрез двоично търсене).
- Ако двете множества от събития имат общи елементи, то в търсения интервал от време [`start`, `end`] залата е заета. В противен случай залата е свободна.

Друго решение, което е по-лесно и по-ефективно, е чрез две инстанции на класа `OrderedBag<T>` от библиотеката "Power Collections for .NET", който има метод за извличане на всички елементи в даден интервал.

8. Тъй като в .NET няма вградена имплементация на структурата от данни приоритетна опашка, можете да използвате структурата `OrderedBag<T>` от Wintellect's Power Collections. За приоритетната опашка (Priority Queue) можете да прочетете повече в съответната статия в Уикипедия: http://en.wikipedia.org/wiki/Priority_Queue.

9. За търсенето по марка, модел и цвят можем да използваме по една хеш-таблица, която търси по даден критерий и връща списък от коли (`Dictionary<string, List<Car>>`).

   За търсенето по година на производство и по ценови диапазон можем да използваме списъци `List<Car>`, сортирани в нарастващ ред съответно по година на производство и по цена.

   Ако търсим по няколко критерия едновременно, можем да извлечем множествата коли по първия критерии, след това множествата коли по втория критерий и т.н. Накрая можем да намерим сечението на множествата. Сечение на две множества се намира, като всеки елемент на по-малкото множество се търси в по-голямото множество. Най-лесно е да се дефинират `Equals()` и `GetHashCode()` за класа `Car` и след това за сечение на множества да се ползва класа `HashSet<Car>`.

Глава 20. Принципи на обектно-ориентираното програмиране

В тази тема...

В настоящата тема ще се запознаем с принципите на обектно-ориентираното програмиране: наследяване на класове и имплементиране на интерфейси, абстракция на данните и поведението, капсулация на данните и скриване на информация за имплементацията на класовете, полиморфизъм и виртуални методи. Ще обясним в детайли принципите за свързаност на отговорностите и функционално обвързване (cohesion и coupling). Ще опишем накратко как се извършва обектно-ориентирано моделиране и как се създава обектен модел по описание на даден бизнес проблем. Ще се запознаем с езика UML и ролята му в процеса на обектно-ориентираното моделиране. Накрая ще разгледаме съвсем накратко концепцията "шаблони за дизайн" и ще дадем няколко типични примера за шаблони, широко използвани в практиката.

Да си припомним: класове и обекти

С класове и обекти се запознахме в главата "Създаване и използване на обекти".

Класовете са описание (модел) на реални предмети или явления, наречени същности (entities). Например класът "Студент".

Класовете имат характеристики – в програмирането са наречени **свойства (properties)**. Например съвкупност от оценки.

Класовете имат и поведение **–** в програмирането са наречени **методи (methods)**. Например явяване на изпит.

Методите и свойствата могат да бъдат видими само в областта на класа, в който са декларирани и наследниците му (private/protected), или видими за всички останали класове (public).

Обектите (objects) са екземпляри (инстанции) на класовете. Например Иван е студент, Петър също е студент.

Обектно-ориентирано програмиране (ООП)

Обектно-ориентираното програмиране е наследник на процедурното (структурно) програмиране. Процедурното програмиране най-общо казано описва програмите чрез група от преизползваеми парчета код (процедури), които дефинират входни и изходни параметри. Процедурните програми представляват съвкупност от процедури, които се извикват една друга.

Проблемът при процедурното програмиране е, че преизползваемостта на кода е трудно постижима и ограничена – само процедурите могат да се преизползват, а те трудно могат да бъдат направени общи и гъвкави. Няма лесен начин да се реализират абстрактни структури от данни, които имат различни имплементации.

Обектно-ориентираният подход залага на парадигмата, че всяка програма работи с данни, описващи същности (предмети и явления) от реалния живот. Например една счетоводна програма работи с фактури, стоки, складове, наличности, продажби и т.н.

Така се появяват обектите – те описват характеристиките (свойства) и поведението (методи) на тези същности от реалния живот.

Основни предимства и цели на ООП – да позволи по-бърза разработка на сложен софтуер и по-лесната му поддръжка. ООП позволява по лесен начин да се преизползва кода, като залага на прости и общоприети правила (принципи). Нека ги разгледаме.

Основни принципи на ООП

За да бъде един програмен език обектно-ориентиран, той трябва не само да позволява работа с класове и обекти, но и трябва да дава възможност за имплементирането и използването на принципите и концепциите на ООП: наследяване, абстракция, капсулация и полиморфизъм. Сега ще разгледаме в детайли всеки от тези основни принципи на ООП.

- **Капсулация (Encapsulation)**

  Ще се научим да скриваме ненужните детайли в нашите класове и да предоставяме прост и ясен интерфейс за работа с тях.

- **Наследяване (Inheritance)**

  Ще обясним как йерархиите от класове подобряват четимостта на кода и позволяват преизползване на функционалност.

- **Абстракция (Abstraction)**

  Ще се научим да виждаме един обект само от гледната точка, която ни интересува, и да игнорираме всички останали детайли.

- **Полиморфизъм (Polymorphism)**

  Ще обясним как да работим по еднакъв начин с различни обекти, които дефинират специфична имплементация на някакво абстрактно поведение.

Наследяване (Inheritance)

Наследяването е основен принцип от обектно-ориентираното програмиране. То позволява на един клас да "наследява" (поведение и характеристики) от друг, по-общ клас. Например лъвът е от семейство котки. Всички котки имат четири лапи, хищници са, преследват жертвите си. Тази функционалност може да се напише веднъж в клас Котка и всички хищници да я преизползват – тигър, пума, рис и т.н.

Как се дефинира наследяване в .NET?

Наследяването в .NET става със специална структура при декларацията на класа. В .NET и други модерни езици за програмиране един клас може да наследи само един друг клас **(single inheritance)**, за разлика от C++, където се поддържа множествено наследяване **(multiple inheritance)**. Ограничението е породено от това, че при наследяване на два класа с еднакъв метод е трудно да се реши кой от тях да се използва (при C++ този проблем е решен много сложно). В .NET могат да се наследяват множество интерфейси, за които ще говорим по-късно.

Класът, който наследяваме, се нарича **клас-родител** или още **базов клас (base class, super class)**.

Наследяване на класове – пример

Да разгледаме един пример за наследяване на класове в .NET. Ето как изглежда базовият (родителски) клас:

Felidae.cs

```
/// <summary>
/// Felidae is latin for "cat"
/// </summary>
public class Felidae
{
   private bool male;

   // This constructor calls another .ctor
   public Felidae() : this(true)
   {}

   // This is the .ctor that is inherited
   public Felidae(bool male)
   {
      this.male = male;
   }

   public bool Male
   {
      get
      {
         return male;
      }
      set
      {
         this.male = value;
      }
   }
}
```

Ето как изглежда и класът-наследник **Lion**:

Lion.cs

```
public class Lion : Felidae
{
   private int weight;

   // Shall be explained in the next paragraph
   public Lion(bool male, int weight) : base(male)
   {
      this.weight = weight;
```

```
  }

  public int Weight
  {
    get
    {
      return weight;
    }
    set
    {
      this.weight = value;
    }
  }
}
```

Ключовата дума base

В горния пример в конструктора на класа **`Lion`** използваме ключовата дума **`base`**. Тя указва да бъде използван базовият клас и позволява достъп до негови методи, конструктори и член-променливи. С **`base()`** можем да извикваме конструктор на базовия клас. С **`base.method(…)`** можем да извикваме метод на базовия клас, да му подаваме параметри и да използваме резултата от него. С **`base.field`** можем да вземем стойността на член-променлива на базовия клас или да й присвоим друга стойност.

В .NET наследените от базовия клас методи, които са декларирани като виртуални (virtual) могат да се **пренаписват** (**override**). Това означава да им се подмени имплементацията, като оригиналният сорс код от базовия клас се игнорира, а на негово място се написва друг код. Повече за пренаписването на методи ще обясним в секцията "Виртуални методи".

Можем да извикваме непренаписан метод от базовия клас и без **`base`**. Употребата на ключовата дума е необходима само ако имаме пренаписан метод или променлива със същото име в наследения клас.

`base` може да се използва изрично, за яснота. `base.method(…)` извиква метод, който задължително е от базовия клас. Такъв код се чете по-лесно, защото знаем къде да търсим въпросния метод.

Имайте предвид, че ситуацията с `this` не е такава. `this` може да означава както метод от конкретния клас, така и метод от който и да е базов клас.

Можете да погледнете примера в секцията нива на достъп при наследяване. В него ясно се вижда до кои членове (методи, конструктори и член-променливи) на базовия клас имаме достъп.

Конструкторите при наследяване

При наследяване на един клас, нашите конструктори задължително трябва да извикат конструктор на базовия клас, за да може и той да инициализира член-променливите си. Ако не го направим изрично, в началото на всеки наш конструктор компилаторът поставя извикване на базовия конструктор без параметри: "`:base()`". Ето и пример:

```
public class ExtendingClass : BaseClass
{
   public ExtendingClass()
}
```

Всъщност изглежда така (намерете разликите ☺):

```
public class ExtendingClass : BaseClass
{
   public ExtendingClass() : base()
}
```

Ако базовият клас няма конструктор по подразбиране (без параметри) или този конструктор е скрит, нашите конструктори трябва да извикат изрично някои от другите конструктори на базовия клас. Липсата на изрично извикване предизвиква грешка при компилация.

Ако един клас има само невидими конструктори (private), то това означава, че той не може да бъде наследяван.

Ако един клас има само невидими конструктори (private), то това означава още много неща – например, че никой не може да създава негови инстанции освен самият той. Всъщност точно по този начин се имплементира един от най-известните шаблони описан накрая на тази глава – нарича се Singleton.

Конструкторите и base – пример

Разгледайте класа **`Lion`** от последния пример, той няма конструктор по подразбиране. Да разгледаме следния клас-наследник на **`Lion`**:

`AfricanLion.cs`

```
public class AfricanLion : Lion
{
   // ...

   // If we comment the next line with ":base(male, weight)"
   // the class will not compile. Try it.
```

```
    public AfricanLion(bool male, int weight)
        : base(male, weight)
    {}

    public override string ToString()
    {
        return string.Format(
            "(AfricanLion, male: {0}, weight: {1})",
            this.Male, this.Weight);
    }

    // ...
}
```

Ако коментираме или изтрием реда "`:base(male, weight);`", класът **`AfricanLion`** няма да се компилира. Опитайте.

Извикването на конструктор на базов клас става извън тялото на конструктора. Идеята е полетата на базовия клас да бъдат инициализирани преди да започнем да инициализираме полета в класа-наследник, защото може те да разчитат на някое поле от базовия клас.

Модификатори на достъп на членове на класа при наследяване

Да си припомним - в главата "Дефиниране на класове" разгледахме основните модификатори на достъпа. За членовете на един клас (методи, свойства, член-променливи) бяха разгледани **`public`**, **`private`**, **`internal`**. Всъщност има още два модификатора - **`protected`** и **`internal protected`**. Ето какво означават те:

- **`protected`** дефинира членове на класа, които са невидими за ползвателите на класа (тези, които го инстанцират и използват), но са видими за класовете наследници
- **`protected internal`** дефинира членове на класа, които са едновременно **`internal`**, тоест видими за ползвателите в цялото асембли, но едновременно с това са и **`protected`** - невидими за ползвателите на класа (извън асемблито), но са видими за класовете наследници (дори и тези извън асемблито).

Когато се наследява един базов клас:

- Всички негови **`public`** и **`protected`**, **`protected internal`** членове (методи, свойства и т.н.) са видими за класа наследник.

- Всички негови **`private`** методи, свойства и член-променливи не са видими за класа наследник.
- Всички негови **`internal`** членове са видими за класа наследник само ако базовият клас и наследникът са в едно и също асембли.

Ето един пример, с който ще демонстрираме нивата на видимост при наследяване:

Felidae.cs

```
/// <summary>
/// Latin word for "cat"
/// </summary>
public class Felidae
{
   private bool male;

   public Felidae() : this(true) {}

   public Felidae(bool male)
   {
      this.male = male;
   }

   public bool Male
   {
      get
      {
         return male;
      }
      set
      {
         this.male = value;
      }
   }
}
```

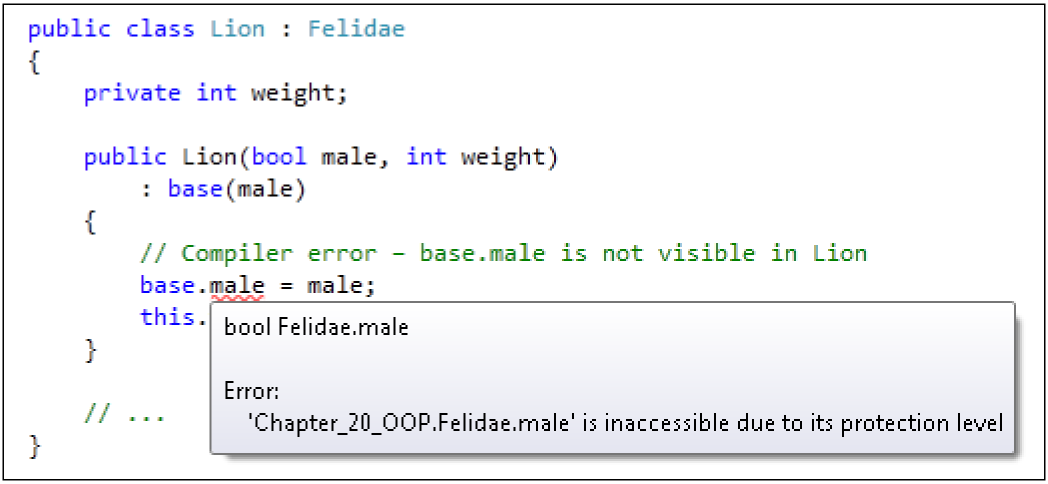
Ето как изглежда и класът **`Lion`**:

Lion.cs

```
public class Lion : Felidae
{
   private int weight;

   public Lion(bool male, int weight)
      : base(male)
   {
      // Compiler error – base.male is not visible in Lion
```

```
        base.male = male;
        this.weight = weight;
    }

    // ...
}
```

Ако се опитаме да компилираме този пример, ще получим грешка, тъй като **`private`** променливата **`male`** от класа **`Felidae`** не е достъпна от класа **`Lion`**:



```
public class Lion : Felidae
{
    private int weight;

    public Lion(bool male, int weight)
        : base(male)
    {
        // Compiler error - base.male is not visible in Lion
        base.male = male;
        this.
    }

    // ...
}
```

```
bool Felidae.male

Error:
  'Chapter_20_OOP.Felidae.male' is inaccessible due to its protection level
```

Класът Object

Появата на обектно-ориентираното програмиране де факто става популярно с езика C++. В него често се налага да се пишат класове, които трябва да работят с обекти от всякакъв тип. В C++ този проблем се решава по начин, който не се смята за много обектно-ориентиран стил (чрез използване на указатели от тип void).

Архитектите на .NET поемат в друга посока. Те създават клас, който всички други класове пряко или косвено да наследяват и до който всеки обект може да бъде преобразуван. В този клас е удобно да бъдат сложени важни методи и тяхната имплементация по подразбиране. Този клас се нарича `Object`.

В .NET всеки клас, който не наследява друг клас изрично, наследява системния клас **`System.Object`** по подразбиране. За това се грижи компилаторът. Всеки клас, който наследява друг клас, наследява индиректно `Object` от него. Така всеки клас явно или неявно наследява `Object` и има в себе си всички негови методи и полета.

Благодарение на това свойство всеки обект може да бъде преобразуван до `Object`. Типичен пример за ползата от неявното наследяване на `Object` е при колекциите, които разгледахме в главите за структури от данни. Списъчните структури (например `System.Collections.ArrayList`) могат да работят с всякакви обекти, защото ги разглеждат като инстанции на класа `Object`.

Специално за колекциите и работата с различни типове обекти има т.нар. Generics (обяснени подробно в главата "Дефиниране на класове"). Тя позволява създаването на типизирани класове – например колекция, която работи само с обекти от тип Lion.

.NET, стандартните библиотеки и Object

В .NET има много предварително написани класове (вече разгледахме доста от тях в главите за колекции, текстови файлове и символни низове). Тези класове са част от .NET платформата – навсякъде, където има .NET, ги има и тях. Тези класове се наричат **обща система от типове – Common Type System (CTS)**.

.NET е една от първите платформи, която идва с такъв богат набор от предварително написани класове. Голяма част от тях работят с `Object`, за да могат да бъдат използвани на възможно най-много места.

В .NET има и доста библиотеки, които могат да се добавят допълнително и съвсем логично се наричат просто клас-библиотеки или още външни библиотеки.

Object, upcasting, downcasting – пример

Нека разгледаме класа `Object` с един пример:

ObjectExample.cs

```
public class ObjectExample
{
   public static void main()
   {
      AfricanLion africanLion = new AfricanLion(true, 80);
      // Implicit casting
      object obj = africanLion;
   }
}
```

В този пример преобразувахме един `AfricanLion` в `Object`. Тази операция се нарича **upcasting** и е позволена, защото `AfricanLion` е непряк наследник на класа `Object`.

Тук е моментът да споменем, че ключовите думи string и object са само компилаторни трикове и всъщност при компилация се заменят съответно със `System.String` и `System.Object`.

Нека продължим примера:

ObjectExample.cs

```
// ...

AfricanLion africanLion = new AfricanLion(true, 80);
// Implicit casting
object obj = africanLion;

try
{
   // Explicit casting
   AfricanLion castedLion = (AfricanLion) obj;
}
catch (InvalidCastException ice)
{
   Console.WriteLine("obj cannot be downcasted to AfricanLion");
}
```

В този пример преобразувахме един **`Object`** в **`AfricanLion`**. Тази операция се нарича **downcasting** и е позволена само ако изрично укажем към кой тип искаме да преминем, защото **`Object`** е родител на **`AfricanLion`** и не е ясно дали променливата **`obj`** е от тип **`AfricanLion`**. Ако не е, се хвърля **`InvalidCastException`**.

Методът Object.ToString()

Един от най-използваните методи, идващи от класа **`Object`**, е **`ToString()`**. Той връща текстово представяне на обекта. Всеки обект има такъв метод и следователно има текстово представяне. Този метод се използва, когато отпечатваме обект чрез **`Console.WriteLine(…)`**.

Object.ToString() – пример

Ето един пример, в който извикваме метода **`ToString()`**:

ToStringExample.cs

```
public class ToStringExample
{
   public static void Main()
   {
        Console.WriteLine(new object());
        Console.WriteLine(new Felidae(true));
        Console.WriteLine(new Lion(true, 80));
    }
}
```

Резултатът е:

```
System.Object
Chapter_20_OOP.Felidae
Chapter_20_OOP.Lion
Press any key to continue . . .
```

Тъй като **`Lion`** не пренаписва (override) метода **`ToString()`**, в конкретния случай се извиква имплементацията от базовия клас. **`Felidae`** също не пренаписва този метод, следователно се извиква имплементацията, наследена от класа **`System.Object`**. В резултата, който виждаме по-горе, се съдържа именното пространство (namespace) на обекта и името на класа.

Пренаписване на ToString() – пример

Нека сега ви покажем колко полезно може да е пренаписването на метода **`ToString()`**, наследено от **`System.Object`**:

AfricanLion.cs

```
public class AfricanLion : Lion
{
    // ...

    public override string ToString()
    {
        return string.Format(
            "(AfricanLion, male: {0}, weight: {1})",
            this.Male, this.Weight);
    }

    // ...
}
```

В горния код използваме **`String.Format(…)`** метода, за да форматираме резултата по подходящ начин. Ето как можем след това да извикваме пренаписания метод **`ToString()`**:

OverrideExample.cs

```
public class OverrideExample
```

```
{
   public static void Main()
   {
      Console.WriteLine(new object());
      Console.WriteLine(new Felidae(true));
      Console.WriteLine(new Lion(true, 80));
      Console.WriteLine(new AfricanLion(true, 80));
   }
}
```

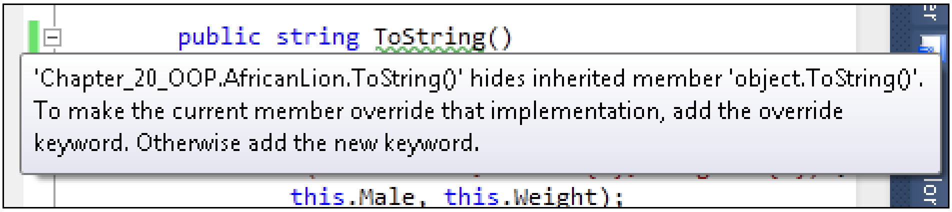
Резултатът е:

```
System.Object
Chapter_20_OOP.Felidae
Chapter_20_OOP.Lion
(AfricanLion, male: True, weight: 80)
Press any key to continue . . .
```

Забележете, че извикването на **`ToString()`** става скрито. Когато на метода **`WriteLine()`** подадем някакъв обект, този обект първо се преобразува до символен низ чрез метода му **`ToString()`** и след това се отпечатва в изходния поток. Така при печатане на конзолата няма нужда изрично да преобразуваме обектите до символен низ.

Виртуални методи и ключовите думи override и new

Трябва да укажем изрично на компилатора, че искаме нашият метод да пренаписва друг. За целта се използва ключовата дума **`override`**. Забележете какво се случва ако я премахнем:



```
public string ToString()
```

'Chapter_20_OOP.AfricanLion.ToString()' hides inherited member 'object.ToString()'. To make the current member override that implementation, add the override keyword. Otherwise add the new keyword.

```
this.Male, this.Weight);
```

Нека си направим един експеримент и използваме ключовата дума **`new`** вместо **`override`**:

```
public class AfricanLion : Lion
{
   // ...

   public new string ToString()
   {

      return string.Format(
        "(AfricanLion, male: {0}, weight: {1})",
        this.Male, this.Weight);
   }

   // ...
}

public class OverrideExample
{
   public static void Main()
   {
      AfricanLion africanLion = new AfricanLion(true, 80);
      string asAfricanLion = africanLion.ToString();
      string asObject = ((object)africanLion).ToString();
      Console.WriteLine( asAfricanLion );
      Console.WriteLine( asObject );
   }
}
```

Резултатът е следният:

```
(AfricanLion, male: True, weight: 80)
Chapter_20_OOP.AfricanLion
Press any key to continue . . .
```

Забелязваме, че когато направим upcast на **AfricanLion** към **object** се извиква имплементацията **Object.ToString()**. Тоест когато използваме ключовата дума **new** създаваме нов метод, който скрива стария и можем да го извикаме само чрез upcast.

Какво става, ако в горния пример върнем думата **override**? Вижте сами резултата:

```
(AfricanLion, male: True, weight: 80)
(AfricanLion, male: True, weight: 80)
Press any key to continue . . .
```

Изненадващо, нали? Оказва се, че когато пренапишем метода (override) дори и с upcast не можем да извикаме старата имплементация. Това е, защото вече не съществуват 2 метода **ToString()** за класа **AfricanLion**, а само един – пренаписан.

Метод, който може да бъде пренаписан, се нарича виртуален метод. В .NET методите по подразбиране не са такива. Ако желаем един метод да може да бъде пренаписан, можем да укажем това с ключовата дума **virtual** в декларацията на метода.

Изричното указване на компилатора, че искаме да пренапишем метод от базов клас (с **override**), е защита против грешки. Ако случайно сбъркаме една буква от името на метода, който се опитваме да пренапишем, или типовете на неговите параметри, компилаторът веднага ще ни съобщи за грешката. Той ще разбере, че нещо не е наред, като не може да намери метод със същата сигнатура в някой от базовите класове.

Виртуалните методи са подробно обяснени в частта, отнасяща се за полиморфизма.

Транзитивност при наследяването

В математиката транзитивност означава прехвърляне на взаимоотношения. Нека вземем операцията "по-голямо". Ако A>B и B>C, то можем да заключим, че A>C. Това означава, че релацията "по-голямо" (>) е транзитивна, защото може еднозначно да бъде определено дали A е по-голямо от C или обратното.

Ако клас **`Lion`** наследява клас **`Felidae`**, а клас **`AfricanLion`** наследява клас **`Lion`**, това индиректно означава, че **`AfricanLion`** наследява **`Felidae`**. Следователно **`AfricanLion`** също има свойство **`Male`**, което е дефинирано във **`Felidae`**. Това полезно свойство позволява определена функционалност да бъде описана в най-подходящия за нея клас.

Транзитивност – пример

Ето един пример, който демонстрира транзитивността при наследяване:

TransitivityExample.cs

```
public class TransitivityExample
{
  public static void Main()
  {
    AfricanLion africanLion = new AfricanLion(true, 15);
    // Property defined in Felidae
    bool male = africanLion.Male;
    africanLion.Male = true;
  }
}
```

Заради транзитивността на наследяването можем да сме сигурни, че всички класове имат **`ToString()`** и другите методи на **`Object`** без значение кой клас наследяват.

Йерархия на наследяване

Ако тръгнем да описваме всички големи котки, рано или късно се стига до сравнително голяма група класове, които се наследяват един друг. Всички тези класове, заедно с базовите такива, образуват йерархия от класове на големите котки. Такива йерархии могат да се опишат най-лесно чрез клас-диаграми. Нека разгледаме какво е това "клас-диаграма".

Клас-диаграми

Клас-диаграмата е един от няколко вида диаграми дефинирани в UML. **UML (Unified Modeling Language)** е нотация за визуализация на различни процеси и обекти, свързани с разработката на софтуер. За UML се говори по-подробно в секцията за нотацията UML. Сега, нека ви разкажем малко за клас-диаграмите, защото те се използват, за да описват визуално йерархиите от класове, наследяването и вътрешността на самите класове.

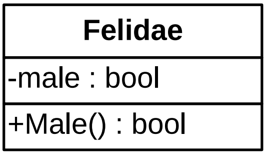
В клас-диаграмите има възприети правила класовете да се рисуват като правоъгълници с име, атрибути (член-променливи) и операции (методи), а връзките между тях се обозначават с различни видове стрелки.

Накратко ще обясним два термина от UML, за по-ясно разбиране на примерите. Единият е **генерализация (generalization)**. Генерализация е обобщаващо понятие за наследяване на клас или имплементация на интерфейс (за интерфейси ще обясним след малко). Другият термин се нарича **асоциация (association)**. Например "Лъвът има лапи", където `Лапа` е друг клас.

Генерализация и асоциация са двата най-основни начина за преизползване на код.

Един клас от клас диаграма – пример

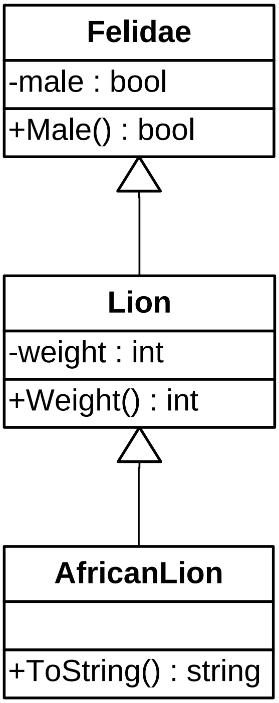
Ето как изглежда една примерна клас-диаграма на един клас:



Класът е представен като правоъгълник, разделен на 3 части, разположени една под друга. В най-горната част е дефинирано името на класа. В следващата част след него са **атрибутите** (термин от UML) на класа (в .NET се наричат член-променливи и свойства). Най-отдолу са **операциите** (в UML) или методите (в .NET). Плюсът/минусът в началото указват дали атрибутът/операцията са видими (**+** означава `public`) или невидими (**-** означава `private`). `Protected` членовете се означават със символа `#`.

Клас-диаграма – генерализация – пример

Ето пример за клас диаграма, показваща генерализация:

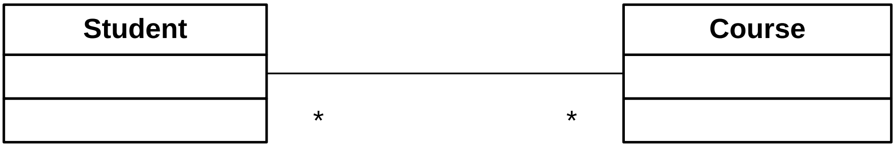

Felidae
-male : bool
+Male() : bool
Lion
-weight : int
+Weight() : int
AfricanLion
+ToString() : string

В този пример стрелките означават генерализация или наследяване.

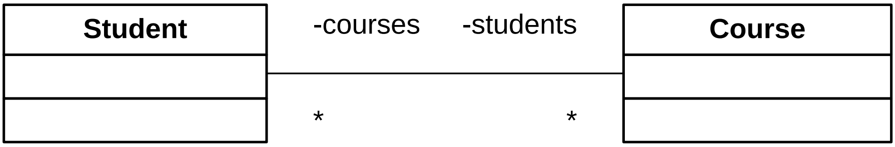
Асоциации

Асоциациите представляват връзки между класовете. Те моделират взаимоотношения. Могат да дефинират множественост (1 към 1, 1 към много, много към 1, 1 към 2, ..., и много към много).

Асоциация **много към много (many-to-many)** се означава по следния начин:

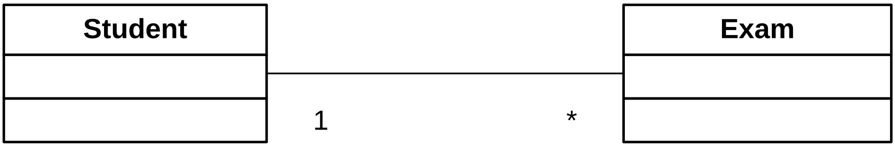

Student
Course
*
*

Асоциация **много към много (many-to-many) по атрибут** се означава по следния начин:

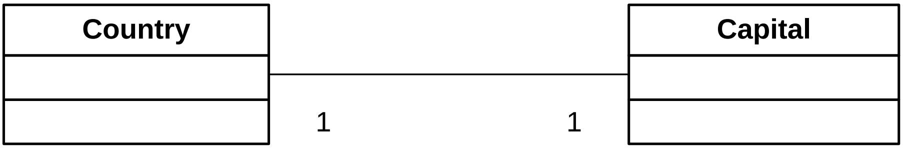

Student
-courses
-students
Course
*
*

В този случай има свързващи атрибути, които показват в кои променливи се държи връзката между класовете.

Асоциация **едно към много (one-to-many)** се означава така:



Асоциация **едно към едно (one-to-one)** се означава така:



От диаграми към класове

От клас-диаграмите най-често се създават класове. Диаграмите улесняват и ускоряват дизайна на класовете на един софтуерен проект.

От горната диаграма можем директно да създадем класове.

Ето класа **`Capital`**:

`Capital.cs`

```
public class Capital { }
```

Ето и класа **`Country`**:

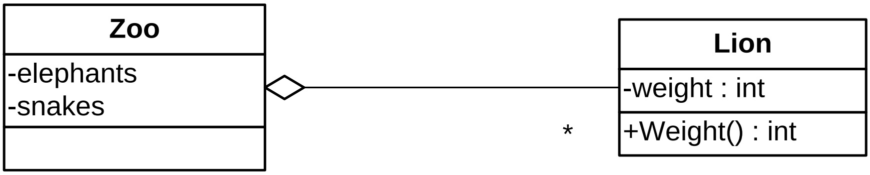
`Country.cs`

```
public class Country
{
  /// <summary>
  /// Country's capital.
  /// </summary>
  private Capital capital;

  // ...

  public Capital Capital
  {
    get
    {
      return capital;
    }
    set
    {
      this.capital = value;
```

```
        }
    }

    // ...
}
```

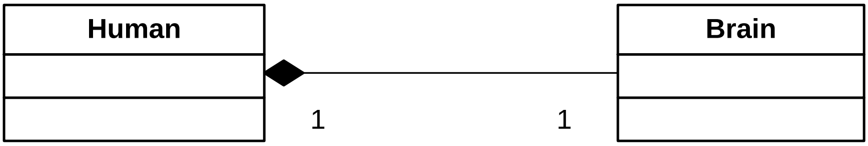
Агрегация

Агрегацията е специален вид асоциация. Тя моделира връзката "цяло / част". **Агрегат** наричаме родителския клас. **Компоненти** наричаме агрегираните класове. В единия край на агрегацията има празен ромб:



Композиция

Запълнен ромб означава композиция. Композицията е агрегация, при която компонентите не могат да съществуват без агрегата (родителя):



Абстракция (Abstraction)

Следващият основен принцип от обектно-ориентираното програмиране, който ще разгледаме, е "абстракция". **Абстракцията** означава да работим с нещо, което знаем как да използваме, но не знаем как работи вътрешно. Например имаме телевизор. Не е нужно да знаем как работи телевизорът отвътре, за да го ползваме. Нужно ни е само дистанционното и с малък брой бутони (интерфейс на дистанционното) можем да гледаме телевизия.

Същото се получава и с обектите в ООП. Ако имаме обект **`Лаптоп`** и той се нуждае от процесор, просто използваме обекта **`Процесор`**. Не знаем (или по-точно не се интересуваме) как той смята вътрешно. За да го използваме, е достатъчно да извикваме метода **`сметни()`** с подходящи параметри.

Абстракцията е нещо, което правим всеки ден. Това е действие, при което игнорираме всички детайли, които не ни интересуват от даден обект, и разглеждаме само детайлите, които имат значение за проблема, който решаваме. Например в хардуера съществува абстракция "устройство за съхранение на данни", което може да бъде твърд диск, USB memory stick, флопи диск или CD-ROM устройство. Всяко от тях работи вътрешно по различен начин, но от гледна точка на операционната система и на програмите в нея те се използват по еднакъв начин – на тях се записват файлове и директории. В Windows имаме Windows Explorer и той умее да работи по еднакъв начин с всички устройства, независимо дали са твърд диск или USB stick. Той работи с абстракцията "устройство за съхранение на данни" (storage device) и не се интересува как точно данните се четат и пишат. За това се грижат драйверите за съответните устройства. Те се явяват конкретни имплементации на интерфейса "устройство за съхранение на данни".

Абстракцията е една от най-важните концепции в програмирането и в ООП. Тя ни позволява да пишем **код, който работи с абстрактни структури от данни** (например списък, речник, множество и други). Имайки абстрактния тип данни, ние можем да работим с него през неговия интерфейс, без да се интересуваме от имплементацията му. Например можем да запазим във файл всички елементи на списък, без да се интересуваме дали той е реализиран с масив, чрез свързан списък или по друг начин. Този код остава непроменен, когато работим с различни конкретни типове данни. Дори можем да пишем нови типове данни (които се появяват на по-късен етап) и те да работят с нашата програма, без да я променяме.

Абстракцията ни позволява и нещо много важно – **да дефинираме интерфейс на нашите програми**, т.е. да дефинираме всички задачи, които тази програма може да извърши, както и съответните входни и изходни данни. Така можем да направим няколко по-малки програми, всяка от които да извършва някаква по-малка задача. Като прибавим това към факта, че можем да работим с абстрактни данни, ни дава голяма гъвкавост при свързването на тези по-малки програми в една по-голяма и ни дава повече възможности за преизползване на код. Тези малки подпрограми се наричат компоненти. Този начин на писане на програми намира широко приложение в практиката, защото ни позволява не само да преизползваме обекти, а дори цели подпрограми.

Абстракция – пример за абстрактни данни

Ето един пример, в който дефинираме конкретен тип данни "африкански лъв", но след това го използваме по абстрактен начин – чрез абстракцията "лъв". Тази абстракция не се интересува от детайлите на всички видове лъвове.

AbstractionExample.cs

```
public class AbstractionExample
{
  public static void Main()
  {
    Lion lion = new Lion(true, 150);
    Felidae bigCat1 = lion;

    AfricanLion africanLion = new AfricanLion(true, 80);
    Felidae bigCat2 = africanLion;
  }
}
```

Интерфейси

В езика C# **интерфейсът** е дефиниция на роля (на група абстрактни действия). Той дефинира какво поведение трябва да има един обект, без да указва как точно се реализира това поведение.

Един обект може да има много роли (да имплементира много интерфейси) и ползвателите му могат да го използват от различни гледни точки.

Например един обект **`Човек`** може да има ролите **`Военен`** (с поведение "стреляй по противника"), **`Съпруг`** (с поведение "обичай жена си"), **`Данъкоплатец`** (с поведение "плати си данъка"). Всеки човек обаче имплементира това поведение по различен начин: **`Иван`** си плаща данъците навреме, **`Георги`** – не навреме, **`Петър`** – въобще не ги плаща.

Някой може да попита защо най-базовият за всички обекти клас `Object` не е всъщност интерфейс. Причината е, че тогава всеки клас щеше да трябва да имплементира една малка, но много важна, група методи, а това би отнемало излишно време. Оказва се, че и не всеки клас има нужда от специфична реализация на `Object.GetHashCode()`, `Object.Equals(…)`, `Object.ToString()`, тоест имплементацията по подразбиране върши работа в повечето случаи. От класа `Object` не е нужно да се пренапише (повторно имплементира) никой метод, но ако се наложи, това може да се направи. Пренаписването на методи е обяснено в детайли в секцията за виртуални методи.

Интерфейси – ключови понятия

В интерфейса може да има само декларации на методи и константи.

Сигнатура на метод (method signature) е съвкупността от името на метода + описание на параметрите (тип и последователност). В един клас/интерфейс всички методи трябва да са с различни сигнатури и да не съвпадат със сигнатури на наследени методи.

Декларация на метод (method declaration) е съвкупността от връщания тип на метода + сигнатурата на метода. Връщаният тип е просто за яснота какво ще върне метода.

⚠	**Това, което идентифицира един метод, е неговата сигнатура. Връщаният тип не е част нея. Причината е, че ако**

два метода се различават само по връщания тип (например два класа, които се наследяват един друг), то не може еднозначно да се идентифицира кой метод трябва да се извика.

Имплементация на клас/метод (class/method implementation) е тялото със сорс код на класа/метода. Най често е заключено между скобите **{** и **}**. При методите се нарича още **тяло на метод**.

Интерфейси – пример

Интерфейсът в .NET се дефинира с ключовата думичка **`interface`**. В него може да има само декларации на методи, както и статични променливи (за константи например). Ето един пример за интерфейс:

Reproducible.cs

```
public interface Reproducible<T> where T:Felidae
{
   T[] Reproduce(T mate);
}
```

За шаблонни типове (Generics) сме говорили в главата "Дефиниране на класове". Интерфейсът, който сме написали, има един метод от тип **`T`** (**`T`** трябва да наследява **`Felidae`**) и връща масив от **`T`**.

Ето как изглежда и класът **`Lion`**, който имплементира интерфейса **`Reproducible`**:

Lion.cs

```
public class Lion : Felidae, Reproducible<Lion>
{
   // ...

   Lion[] Reproducible<Lion>.Reproduce(Lion mate)
   {
      return new Lion[]{new Lion(true, 12), new Lion(false, 10)};
   }
}
```

Името на интерфейса се записва в декларацията на класа (първия ред) и се специфицира шаблонният клас.

Можем да укажем метод на кой интерфейс имплементираме, като му напишем името:

```
Lion[] Reproducible<Lion>.Reproduce(Lion mate)
```

В интерфейса методите само се декларират, имплементацията е в класа, който имплементира интерфейса – `Lion`.

Класът, който имплементира даден интерфейс, трябва да имплементира всеки метод от него. Изключение – ако класът е абстрактен, тогава може да имплементира нула, няколко или всички методи. Всички останали методи се имплементират в някой от класовете наследници.

Абстракция и интерфейси

Най-добрият начин да се реализира абстракция е да се работи с интерфейси. Един компонент работи с интерфейси, които друг имплементира. Така подмяната на втория компонент няма да се отрази на първия, стига новият компонент да имплементира старите интерфейси. Интерфейсът се нарича още **договор** (**contract**). Всеки компонент, имплементирайки един интерфейс, спазва определен договор (сигнатурата на методите). Така два компонента, стига да спазват правилата на договора, могат да общуват един с друг, без да знаят как работи другата страна.

Примери за важни интерфейси от Common Type System (CTS) са **`System.Collections.Generic.IList<T>`** и **`System.Collections.Generic.ICollection<T>`**. Всички стандартни колекции имплементират тези интерфейси и различните компоненти си прехвърлят различни имплементации (масиви или свързани списъци, хеш-таблици, червено-черни дървета и др.) винаги под общ интерфейс.

Колекциите са един отличен пример на обектно-ориентирана библиотека с класове и интерфейси, при която се използват много активно всички основни принципи на ООП: абстракция, наследяване, капсулация и полиморфизъм.

Кога да използваме абстракция и интерфейси?

Отговорът на този въпрос е: винаги, когато искаме да постигнем абстракция на данни или действия, чиято имплементация по-късно може да се подмени. Код, който комуникира с друг код чрез интерфейси е много по-издръжлив срещу промени, отколкото код, написан срещу конкретни класове. Работата през интерфейси е често срещана и силно препоръчвана практика – едно от основните правила за писане на качествен код.

Кога да пишем интерфейси?

Винаги е добра идея да се използват интерфейси, когато се предоставя функционалност на друг компонент. В интерфейса се слага само функционалността (като декларация), която другите трябва да виждат.

Вътрешно в една програма/компонент интерфейсите могат да се използват за дефиниране на роли. Така един обект може да се използва от много класове чрез различните му роли.

Капсулация (Encapsulation)

Капсулацията е един от основните принципи на обектно-ориентираното програмиране. Тя се нарича още "скриване на информацията" (**information hiding**). Един обект трябва да предоставя на ползвателя си само необходимите средства за управление. Една **Секретарка** ползваща един **Лаптоп** знае само за екран, клавиатура и мишка, а всичко останало е скрито. Тя няма нужда да знае за вътрешността на **Лаптопа**, защото не й е нужно и може да оплеска нещо. Тогава част от свойствата и методите остават скрити за нея.

Изборът какво е скрито и какво е публично видимо е на този, който пише класа. Когато програмираме, трябва да дефинираме като **private** (скрит) всеки метод или поле, които не искаме да се ползват от друг клас.

Капсулация – примери

Ето един пример за скриване на методи, които не е нужно да са известни на потребителя, а се ползват вътрешно само от автора на класа. Първо дефинираме абстрактен клас **Felidae**, който дефинира публичните операции на котките (независимо какви точно котки имаме):

Felidae.cs

```
public class Felidae
{
    public virtual void Walk()
    {
        // ...
    }

    // ...
}
```

Ето как изглежда класът **Lion**:

Lion.cs

```
public class Lion : Felidae, Reproducible<Lion>
{
    // ...

    private Paw frontLeft;
    private Paw frontRight;
    private Paw bottomLeft;
    private Paw bottomRight;

    private void MovePaw(Paw paw) {
      // ...
   }

   public override void Walk()
   {
      this.movePaw(frontLeft);
      this.movePaw(frontRight);
      this.movePaw(bottomLeft);
      this.movePaw(bottomRight);
   }

   // ...
}
```

Публичният метод **`Walk()`** извиква 4 пъти някакъв друг скрит (**`private`**) метод. Така базовият клас е кратък – само един метод. Имплементацията обаче извиква друг метод, също част от имплементацията, но скрит за ползвателя на класа. Така класът **`Lion`** не разкрива публично информация за това как работи вътрешно и това му дава възможност на по-късен етап да промени имплементацията си без останалите класове да разберат (и да имат нужда от промяна).

Полиморфизъм (Polymorphism)

Следващият основен принцип от обектно-ориентираното програмиране е "полиморфизъм". **Полиморфизмът** позволява третирането на обекти от наследен клас като обекти от негов базов клас. Например големите котки (базов клас) хващат жертвите си (метод) по различен начин. Лъвът (клас наследник) ги дебне, докато Гепардът (друг клас-наследник) просто ги надбягва.

Полиморфизмът дава възможността да третираме произволна голяма котка просто като голяма котка и да кажем "хвани жертвата си", без значение каква точно е голямата котка.

Полиморфизмът може много да напомня на абстракцията, но в програмирането се свързва най-вече с пренаписването (override) на методи в наследените класове с цел промяна на оригиналното им поведение, наследено от базовия клас. Абстракцията се свързва със създаването на интерфейс на компонент или функционалност (дефиниране на роля). Пренаписването на методи ще разгледаме в детайли след малко.

Абстрактни класове

Какво става, ако искаме да кажем, че класът **`Felidae`** е непълен и само наследниците му могат да имат инстанции? Това става с ключовата дума **`abstract`** пред името на класа и означава, че класът не е готов и не може да бъде инстанциран. Такъв клас се нарича **абстрактен клас**. А как да укажем коя точно част от класа не е пълна? Това отново става с ключовата дума **`abstract`** пред името на метода, който трябва да бъде имплементиран. Този метод се нарича **абстрактен метод** и не може да притежава имплементация, а само декларация.

Всеки клас, който има поне един абстрактен метод, трябва да бъде абстрактен. Логично, нали? Обратното, обаче не е в сила. Възможно е да дефинираме клас като абстрактен дори когато в него няма нито един абстрактен метод.

Абстрактните класове са нещо средно между клас и интерфейс. Те могат да дефинират обикновени методи и абстрактни методи. Обикновените методи имат тяло (имплементация), докато абстрактните методи са празни (без имплементация) и са оставени да бъдат реализирани от класовете-наследници.

Абстрактен клас – примери

Да разгледаме един пример за абстрактен клас:

Felidae.cs

```
/// <summary>
/// Latin word for "cat"
/// </summary>
public abstract class Felidae
{
   // ...

   protected void Hide()
   {
      // ...
   }

   protected void Run()
   {
      // ...
   }

   public abstract bool CatchPray(object pray);
}
```

Забележете в горния пример как нормалните методи **`Hide()`** и **`Run()`** имат тяло, а абстрактният метод **`CatchPray()`** няма тяло. Забележете, че методите са **`protected`**.

Ето как изглежда имплементацията:

Lion.cs

```
public class Lion : Felidae, Reproducible<Lion>
{
   protected void Ambush()
   {
      // ...
   }

   public override bool CatchPray(object pray)
   {
      base.Hide();
      this.Ambush();
      base.Run();
      // ...
      return false;
   }
}
```

Ето още един пример за абстрактно поведение, реализирано чрез абстрактен клас и полиморфно извикване на абстрактен метод. Първо дефинираме абстрактния клас **`Animal`**:

Animal.cs

```
public abstract class Animal
{
   public void PrintInformation()
   {
      Console.WriteLine("I am {0}.", this.GetType().Name);
      Console.WriteLine(GetTypicalSound());
   }

   protected abstract String GetTypicalSound();
}
```

Дефинираме и класа **`Cat`**, който наследява абстрактния клас **`Animal`** и дефинира имплементация за абстрактния метод **`GetTypicalSound()`**:

Cat.cs

```
public class Cat : Animal
{
   protected override String GetTypicalSound()
   {
      return "Miaoooow!";
   }
}
```

Ако изпълним следната програма:

```
public class AbstractClassExample
{
   public static void Main()
   {
      Animal cat = new Cat();
      cat.PrintInformation();
   }
}
```

... ще получим следния резултат:

```
I am Cat.
Miaoooow!
Press any key to continue . . .
```

В примера методът **`PrintInformation()`** от абстрактния клас свършва своята работа като разчита на резултата от извикването на абстрактния метод **`GetTypicalSound()`**, който се очаква да бъде имплементиран по различен начин за различните животни (различните наследници на класа **`Animal`**). Различните животни издават различни звуци, но отпечатването на информация за животно е една и съща функционалност за всички животни и затова е изнесена в базовия клас.

Чист абстрактен клас

Абстрактните класове, както и интерфейсите не могат да се инстанцират. Ако се опитате да създадете инстанция на абстрактен клас, ще получите грешка по време на компилация.

Понякога даден клас може да бъде деклариран като абстрактен дори и да няма нито един абстрактен метод, просто, за да се забрани директното му използване, без да се създава инстанция на негов наследник.

Чист абстрактен клас (pure abstract class) е абстрактен клас, който няма нито един имплементиран метод, както и нито една член променлива. Много напомня на интерфейс. Основната разлика е, че един клас може да имплементира много интерфейси и наследява само един клас (бил той и чист абстрактен клас).

В началото при съществуването на множествено наследяване не е имало нужда от интерфейси. За да бъде заместено, се е наложило да се появят интерфейсите, които да носят многото роли на един обект.

Виртуални методи

Метод, който може да се пренапише в клас наследник, се нарича **виртуален метод (virtual method)**. Методите в .NET не са виртуални по подразбиране. Ако искаме да бъдат виртуални, ги маркираме с ключовата дума **`virtual`**. Тогава клас-наследник може да декларира и дефинира метод със същата сигнатура.

Виртуалните методи са важни за **пренаписването на методи (method overriding)**, което е в сърцето на полиморфизма.

Виртуални методи – пример

Имаме клас, наследяващ друг, като и двата имат общ метод. И двата метода пишат на конзолата. Ето как изглежда класът **`Lion`**:

Lion.cs

```
public class Lion : Felidae, Reproducible<Lion>
{
   public override void CatchPray(object pray)
   {
      Console.WriteLine("Lion.CatchPray");
   }
}
```

Ето как изглежда и класът **`AfricanLion`**:

AfricanLion.cs

```
public class AfricanLion : Lion
{
   public override void CatchPray(object pray)
   {
      Console.WriteLine("AfricanLion.CatchPray");
   }
}
```

Правим три опита за създаване на инстанции и извикване на метода **`CatchPray`**.

VirtualMethodsExample.cs

```
public class VirtualMethodsExample
{
   public static void Main()
   {
      {
         Lion lion = new Lion(true, 80);
```

```
        lion.CatchPray(null);
        // Will print "Lion.CatchPray"
      }

      {
        AfricanLion lion = new AfricanLion(true, 120);
        lion.CatchPray(null);
        // Will print "AfricanLion.CatchPray"
      }

      {
        Lion lion = new AfricanLion(false, 60);
        lion.CatchPray(null);
        // Will print "AfricanLion.CatchPray", because
        // the variable lion has value of type AfricanLion
      }
    }
}
```

В последния опит ясно се вижда как всъщност се извиква пренаписаният метод, а не базовият. Това се случва, защото се проверява кой всъщност е истинският клас, стоящ зад променливата, и се проверява дали той има имплементиран (пренаписан) този метод.

Пренаписването на методи се нарича още: **препокриване (подмяна) на виртуален метод**.

Както виртуалните, така и абстрактните методи могат да бъдат препокривани. Абстрактните методи всъщност представляват виртуални методи без конкретна имплементация. Всички методи, които са дефинирани в даден интерфейс са абстрактни и следователно виртуални, макар и това да не е дефинирано изрично.

Виртуални методи и скриване на методи

В горния пример имплементацията на базовия клас остана скрита и неизползвана. Ето как можем да ползваме и нея като част от новата имплементация (в случай че не искаме да подменим, а само да допълним старата имплементация).

Ето как изглежда и класът **`AfricanLion`**:

`AfricanLion.cs`

```
public class AfricanLion : Lion
{
  public override void CatchPray(object pray)
  {
    Console.WriteLine("AfricanLion.CatchPray");
    Console.WriteLine("calling base.CatchPray");
```

```
        Console.Write("\t");
        base.CatchPray(pray);
        Console.WriteLine("...end of call.");
    }
}
```

В този пример при извикването на **AfricanLion.catchPray(…)** ще се изпишат 3 реда на конзолата:

```
AfricanLion.CatchPray
calling base.CatchPray
        Lion.CatchPray
...end of call.
```

Разликата между виртуални и невиртуални методи

Някой може да попита каква е разликата между виртуалните и невиртуалните методи.

Виртуални методи се използват, когато очакваме наследяващите класове да променят/допълват/изменят дадена функционалност. Например методът **Object.ToString()** позволява наследяващите класове да променят както си искат имплементацията. И тогава дори когато работим с един обект не директно, а чрез upcast до object пак използваме пренаписаната имплементация на виртуалните методи.

Виртуалните методи са ключова способност на обектите когато говорим за абстракция и работа с абстрактни типове.

Запечатването на методи (**sealed**) се прави, когато разчитаме на дадена функционалност и не желаем тя да бъде променяна. Разбрахме, че методите по принцип са запечатани. Но ако искаме един виртуален метод от базов клас да запечатаме в класа наследник, използваме **override sealed**.

Класът **string** няма нито един виртуален метод. Всъщност наследяването на **string** е забранено изцяло с ключовата дума **sealed** в декларацията на класа. Ето част от декларацията на **string** и **object** (триеточието в квадратните скоби указва пропуснат код, който не е релевантен):

```
namespace System
{
  [...] public class Object
  {
    [...] public Object();

    [...] public virtual bool Equals(object obj);
    [...] public static bool Equals(object objA, object objB);
    [...] public virtual int GetHashCode();
    [...] public Type GetType();
    [...] protected object MemberwiseClone();
    [...] public virtual string ToString();
  }


  [...] public sealed class String : [...]
  {
    [...] public String(char* value);

    [...] public int IndexOf(string value);
    [...] public string Normalize();
    [...] public string[] Split(params char[] separator);
    [...] public string Substring(int startIndex);
    [...] public string ToLower(CultureInfo culture);

    [...]
  }
}
```

Кога да използваме полиморфизъм?

Отговорът на този въпрос е прост: винаги, когато искаме да предоставим възможност имплементацията на даден метод да бъде подменен в клас-наследник. Добро правило е да се работи с възможно най-базовия клас или направо с интерфейс. Така промените върху използваните класове се отразяват в много по-малка степен върху класовете, които ние пишем. Колкото по-малко знае една програма за обкръжаващите я класове, толкова по-малко промени (ако въобще има някакви) трябва да претърпи тя.

Свързаност на отговорностите и функционално обвързване (cohesion и coupling)

Термините cohesion и coupling са неразривно свързани с ООП. Те допълват и дообясняват някои от принципите, които описахме до момента. Нека се запознаем с тях.

Свързаност на отговорностите (cohesion)

Понятието **cohesion (свързаност на отговорностите)** показва до каква степен различните задачи и отговорности на една програма или един компонент са свързани помежду си, т.е. колко фокусирана е програмата в решаването на една единствена задача. Разделя се на **силна свързаност (strong cohesion)** и **слаба свързаност (weak cohesion)**.

Силна свързаност на отговорностите (strong cohesion)

Когато кохезията (cohesion) е силна, това показва, че отговорностите и задачите на една единица код (метод, клас, компонент, подпрограма) са свързани помежду си и се стремят да решат общ проблем. Това е нещо, към което винаги трябва да се стремим. Strong cohesion е типична характеристика на висококачествения софтуер.

Силна свързаност за клас

Силна свързаност на отговорностите (strong cohesion) в един клас означава, че този клас описва само един субект. По-горе споменахме, че един субект може да има много роли (Петър е военен, съпруг, данъкоплатец). Всички тези роли се описват в един и същ клас. Силната свързаност означава, че класът решава една задача, един проблем, а не много едновременно. Клас, който прави много неща едновременно, е труден за разбиране и поддръжка. Представете си клас, който реализира едновременно хеш-таблица, предоставя функции за печатане на принтер, за пращане на e-mail и за работа с тригонометрични функции. Какво име ще дадем на този клас? Ако се затрудняваме в отговора на този въпрос, това означава, че нямаме силна свързаност на отговорностите (cohesion) и трябва да разделим класа на няколко по-малки, всеки от които решава само една задача.

Силна свързаност за клас – пример

Като пример за силна свързаност на отговорности можем да дадем класа **`System.Math`**. Той изпълнява една единствена задача – предоставя математически изчисления и константи:

- **`Sin()`, `Cos()`, `Asin()`**
- **`Sqrt()`, `Pow()`, `Exp()`**
- **`Math.PI`, `Math.E`**

Силна свързаност за метод

Един метод е добре написан, когато изпълнява само една задача и я изпълнява добре. Метод, който прави много неща, свързани със съвсем различни задачи, има лоша кохезия и трябва да се раздели на няколко по-прости метода, които решават само една задача. И тук стои въпросът какво име ще дадем на метод, който търси прости числа, чертае 3D графика на екрана, комуникира по мрежата и печата на принтер справки, извлечени от база данни. Такъв метод има лоша кохезия и трябва да се раздели логически на няколко метода.

Слаба свързаност на отговорностите (weak cohesion)

Слаба свързаност се наблюдава при методи, които вършат по няколко задачи. Тези методи трябва да приемат няколко различни групи параметри, за да извършат различните задачи. Понякога това налага несвързани логически данни да се обединяват за точно такива методи. Използването на слаба кохезия (weak cohesion) е вредно и трябва да се избягва!

Слаба свързаност на отговорностите – пример

Ето един пример за клас, който има слаба свързаност на отговорностите (weak cohesion):

```
public class Magic
{
  public void PrintDocument(Document d) { ... }
  public void SendEmail(string recipient,
    string subject, string text) { ... }
  public void CalculateDistanceBetweenPoints(
    int x1, int y1, int x2, int y2) { ... }
}
```

Добри практики за свързаност на отговорностите

Съвсем логично силната свързаност е "добрият" начин на писане на код. Понятието се свързва с по-прост и по-ясен сорс код – код, който по-лесно се поддържа и по-лесно се преизползва (поради по-малкия на брой задачи, които той изпълнява).

Обратно, при слаба свързаност всяка промяна е бомба със закъснител, защото може да засегне друга функционалност. Понякога една логическа задача се разпростира върху няколко модула и така промяната ѝ е по-трудоемка. Преизползването на код също е трудно, защото един компонент върши няколко несвързани задачи и за да се използва отново, трябва да са на лице точно същите условия, което трудно може да се постигне.

Функционално обвързване (coupling)

Функционално обвързване (coupling) описва най-вече до каква степен компонентите / класовете зависят един от друг. Дели се на **функционална независимост (loose coupling)** и **силна взаимосвързаност (tight coupling)**. Функционалната независимост обикновено идва заедно със слабата свързаност на отговорностите и обратно.

Функционална независимост (loose coupling)

Функционалната независимост (loose coupling) се характеризира с това, че единиците код (подпрограма / клас / компонент) общуват с други такива през ясно дефинирани интерфейси (договори) и промяната в имплементацията на един компонент не се отразява на другите, с които той общува. Когато пишете програмен код, не трябва да разчитате на вътрешни характеристики на компонентите (специфично поведение, неописано в интерфейсите).

Договорът трябва да е максимално опростен и да дефинира единствено нужните за работата на този компонент поведения, като скрива всички ненужни детайли.

Функционалната независимост е характеристика на кода, към която трябва да се стремите. Тя е една от отличителните черти на качествения програмен код.

Loose coupling – пример

Ето един пример, в който имаме функционална независимост между класовете и методите:

```
class Report
{
  public bool LoadFromFile(string fileName) {…}
  public bool SaveToFile(string fileName) {…}
}

class Printer
{
  public static int Print(Report report) {…}
}

class Example
{
  public static void Main()
  {
    Report myReport = new Report();
    myReport.LoadFromFile("DailyReport.xml");
    Printer.Print(myReport);
  }
}
```

В този пример никой клас и никой метод не зависи от останалите. Методите зависят само от параметрите, които им се подават. Ако някой метод ни потрябва в следващ проект, лесно ще можем да го извадим и използваме отново.

Силна взаимосвързаност (tight coupling)

Силна взаимосвързаност имаме при много входни параметри и изходни параметри и при използване на неописани (в договора) характеристики на друг компонент (например зависимост от статични полета в друг клас). При използване на много т. нар. контролни променливи, които оказват какво да е поведението със същинските данни. Силната взаимосвързаност между два или повече метода, класа или компонента означава, че те не могат да работят независимо един от друг и че промяната в един от тях ще засегне и останалите. Това води до труден за четене код и големи проблеми при поддръжката му.

Tight coupling – пример

Ето един пример, в който имаме силна взаимосвързаност между класовете и методите:

```
class MathParams
{
  public static double operand;
  public static double result;
}

class MathUtil
{
  public static void Sqrt()
  {
    MathParams.result = CalcSqrt(MathParams.operand);
  }
}

class SpaceShuttle
{
  public static void Main()
  {
    MathParams.operand = 64;
    MathUtil.Sqrt();
    Console.WriteLine(MathParams.result);
  }
}
```

Такъв код е труден за разбиране и за поддръжка, а възможността за грешки при използването му е огромна. Помислете какво се случва, ако друг метод, който извиква **`Sqrt()`**, подава параметрите си през същите статични променливи **`operand`** и **`result`**.

Ако се наложи в следващ проект да използваме същата функционалност за извличане на корен квадратен, няма да можем просто да си копираме метода **`Sqrt()`**, а ще трябва да копираме класовете **`MathParams`** и **`MathUtil`** заедно с всичките им методи. Това прави кода труден за преизползване.

Всъщност горният код е пример за лош код по всички правила на процедурното и обектно-ориентираното програмиране и ако се замислите, сигурно ще се сетите за още поне няколко неспазени препоръки, които сме ви давали до момента.

Добри практики за функционално обвързване

Най-честият и препоръчителен начин за извикване на функционалност на един добре написан модул е през интерфейси, така функционалността може да се подменя, без клиентите на този код да трябва да се променят. Жаргонният израз за това е "програмиране срещу интерфейси".

Интерфейсът най-често описва "договора", който този модул спазва. Добрата практика е да не се разчита на нищо повече от описаното в този договор. Използването на вътрешни класове, които не са част от публичния интерфейс на един модул не е препоръчително, защото тяхната имплементация може да се подмени без това да подмени договора (за това вече споменахме в секцията "Абстракция").

Добра практика е методите да са гъвкави и да са готови да работят с всички компоненти, които спазват интерфейса им, а не само с определени такива (тоест да имат неявни изисквания). Последното би означавало, че тези методи очакват нещо специфично от компонентите, с които могат да работят. Добра практика е също всички зависимости да са ясно описани и видими. Иначе поддръжката на такъв код става трудна (пълно е с подводни камъни).

Добър пример за strong cohesion и loose coupling са класовете в `System.Collections` и `System.Collections.Generic`. Класовете за работа с колекции имат силна кохезия. Всеки от тях решава една задача и позволява лесна преизползваемост. Тези класове притежават и другата характеристика на качествения програмен код: loose coupling. Класовете, реализиращи колекциите, са необвързани един с друг. Всеки от тях работи през строго дефиниран интерфейс и не издава детайли за своята имплементация. Всички методи и полета, които не са от интерфейса, са скрити, за да се намали възможността за обвързване на други класове с тях. Методите в класовете за колекции не зависят от статични променливи и не разчитат на никакви входни данни, освен вътрешното си състояние и подадените им параметри. Това е добрата практика, до която рано или късно всеки програмист достига като понатрупа опит.

Код като спагети (spaghetti code)

Спагети код е неструктуриран код с неясна логика, труден за четене, разбиране и за поддържане. Това е код, в който последователността е нарушена и объркана. Това е код, който има weak cohesion и tight coupling. Този код се свързва се със спагети, защото също като тях е оплетен и завъртян. Като дръпнеш един спагет (т.е. един клас или метод), цялата чиния спагети може да се окаже, оплетена в него (т. е. промяна на един метод или клас води до още десетки други промени поради силната зависимост между тях). Спагети кодът е почти невъзможно да се преизползва, защото няма как да отделиш тази част от него, която върши работа.

void*
Realocate
{ void*t
temp = ma
memcpy((v
free(buf)
buf = mal
memset(b
memcpy((
return b
}

Спагети кодът се получава, когато сте писали някакъв код, след това сте го допълнили, след това изискванията са се променили и вие сте нагодили кода към тях, след това пак са се пременили и т.н. С времето спагетите се оплитат все повече и повече и идва момент, в който всичко трябва да се пренапише от нулата.

Cohesion и coupling в инженерните дисциплини

Ако си мислите, че принципите за strong cohesion и loose coupling се отнасят само за програмирането, дълбоко се заблуждавате. Това са здрави инженерни принципи, които ще срещнете в строителството, в машиностроенето, в електрониката и на още хиляди места.

Да вземем за пример един твърд диск:

Той решава една единствена задача, нали? Твърдият диск решава задачата за съхранение на данни. Той не охлажда компютъра, не издава звуци, няма изчислителна сила и не се ползва като клавиатура. Той е свързан с компютъра само с 2 кабела, т.е. има прост интерфейс за достъп и не е обвързан с другите периферни устройства. Твърдият диск работи самостоятелно и другите устройства не се интересуват от това точно как работи. Централния процесор му казва "чети" и той чете, след това му казва "пиши" и той пише. Как точно го прави е скрито вътре в него. Различните модели могат да работят по различен начин, но това си е техен проблем. Виждате, че един твърд диск притежава strong cohesion, loose coupling, добра абстракция и добра капсулация. Така трябва да реализирате и вашите класове – да вършат една задача, да я вършат добре, да се обвързват минимално с другите класове (или въобще да не се обвързват, когато е възможно), да имат ясен интерфейс и добра абстракция и да скриват детайлите за вътрешната си работа.

Ето един друг пример: представете си какво щеше да стане, ако на дънната платка на компютъра бяха запоени процесорът, твърдият диск, CD-ROM устройството и клавиатурата. Това означава, че като ви се повреди някой клавиш от клавиатурата, ще трябва да изхвърлите на боклука целия компютър. Виждате, че при tight coupling и weak cohesion хардуерът не може да работи добре. Същото се отнася и за софтуера.

Обектно-ориентирано моделиране (ООМ)

Нека приемем, че имаме да решаваме определен проблем или задача. Този проблем идва обикновено от реалния свят. Той съществува в дадена реалност, която ще наричаме заобикаляща го среда.

Обектно-ориентираното моделиране (ООМ) е процес, свързан с ООП, при който се изваждат всички обекти, свързани с проблема, който решаваме (създава се модел). Изваждат се само тези техни характеристики, които са свързани с решаването на конкретния проблем. Останалите се игнорират. Така вече си създаваме нова реалност, която е опростена версия на оригиналната (неин модел), и то такава, че ни позволява да си решим проблема или задачата.

Например, ако моделираме система за продажба на билети, за един пътник важни характеристики биха могли да бъдат неговото име, неговата възраст, дали ползва намаление и дали е мъж, или жена (ако продаваме спални места). Пътникът има много други характеристики, които не ни интересуват, примерно какъв цвят са му очите, кой номер обувки носи, какви книги харесва или каква бира пие.

При моделирането се създава опростен модел на реалността с цел решаване на конкретната задача. При обектно-ориентираното моделиране моделът се прави със средствата на ООП: чрез класове, атрибути на класовете, методи в класовете, обекти, взаимоотношения между класовете и т.н. Нека разгледаме този процес в детайли.

Стъпки при обектно-ориентираното моделиране

Обектно-ориентираното моделиране обикновено се извършва в следните стъпки:

- Идентификация на класовете.
- Идентификация на атрибутите на класовете.
- Идентификация на операциите върху класовете.
- Идентификация на връзките между класовете.

Ще разгледаме кратък пример, с който ще ви покажем как могат да се приложат тези стъпки.

Идентификация на класовете

Нека имаме следната извадка от заданието за дадена система:

> На потребителя трябва да му е позволено да описва всеки продукт по основните му характеристики, включващи име и номер на продукта. Ако бар-кодът не съвпада с продукта, тогава трябва да бъде генерирана грешка на екрана за съобщения. Трябва да има дневен отчет за всички

транзакции, специфицирани в секция 9.

Ето как идентифицираме ключовите понятия:

На **потребителя** трябва да му е позволено да описва всеки **продукт** по основните му **характеристики**, включващи **име** и **номер на продукта**. Ако **бар-кодът** не съвпада с продукта, тогава трябва да бъде генерирана **грешка** на **екрана за съобщения**. Трябва да има **дневен отчет** за всички **транзакции**, специфицирани в секция 9.

Току-що идентифицирахме класовете, които ще ни трябват. Имената на класовете са съществителните имена в текста, най-често нарицателни в единствено число, например `Студент`, `Съобщение`, `Лъв`. Избягвайте имена, които не идват от текста, примерно: `СтраненКлас`, `АдресКойтоИмаСтудент`.

Понякога е трудно да се прецени дали някой предмет или явление от реалния свят трябва да бъде клас. Например адресът може да е клас `Address` или символен низ. Колкото по-добре проучим проблема, толкова по-лесно ще решим кое трябва да е клас. Когато даден клас стане прекалено голям и сложен, той трябва да се декомпозира на няколко по-малки класове.

Идентификация на атрибутите на класовете

Класовете имат атрибути (характеристики), например: класът `Student` има име, учебно заведение и списък от курсове. Не всички характеристики са важни за софтуерната система. Например: за класа `Student` цветът на очите е несъществена характеристика. Само съществените характеристики трябва да бъдат моделирани.

Идентификация на операциите върху класовете

Всеки клас трябва да има ясно дефинирани отговорности – какви обекти или процеси от реалния свят представя, какви задачи изпълнява. Всяко действие в програмата се извършва от един или няколко метода в някой клас. Действията се моделират с операции (методи).

За имената на методите се използват глагол + съществително. Примери: `PrintReport()`, `ConnectToDatabase()`. Не може веднага да се дефинират всички методи на даден клас. Дефинираме първо най-важните методи – тези, които реализират основните отговорности на класа. С времето се появяват още допълнителни методи.

Идентификация на връзките между класовете

Ако един студент е от определен факултет и за задачата, която решаваме, това е важно, тогава студент и факултет са свързани. Тоест класът

`Факултет` има списък от `Студенти`. Тези връзки наричаме още асоциации (спомнете си секцията "клас-диаграми").

Нотацията UML

UML (Unified Modeling Language) бе споменат в секцията за наследяване. Там разгледахме клас-диаграмите. UML нотацията дефинира още няколко вида диаграми. Нека разгледаме накратко някои от тях.

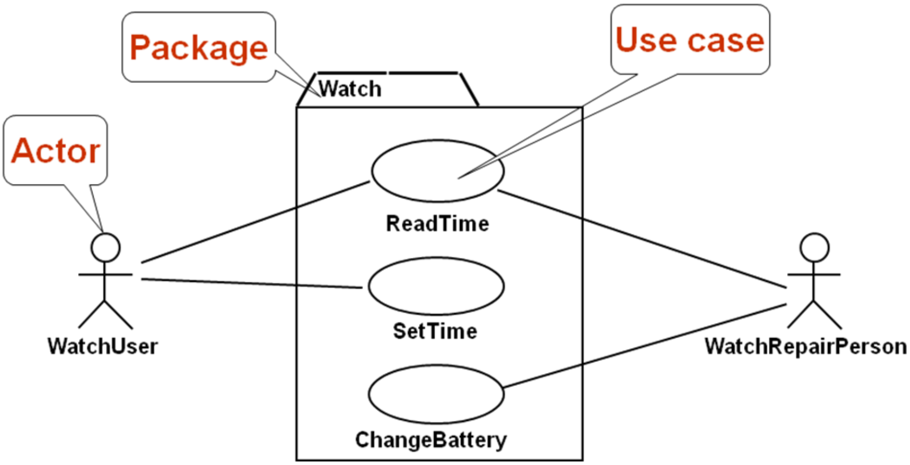
Use case диаграми (случаи на употреба)

Използват се при извличане на изискванията за описание на възможните действия. **Актьорите (actors)** представят роли (типове потребители).

Случаите на употреба (use cases) описват взаимодействие между актьорите и системата. Use case моделът е група use cases – предоставя пълно описание на функционалността на системата.

Use case диаграми – пример

Ето как изглежда една use case диаграма:



Актьорът е някой, който взаимодейства със системата (потребител, външна система или примерно външната среда). Актьорът има уникално име и евентуално описание.

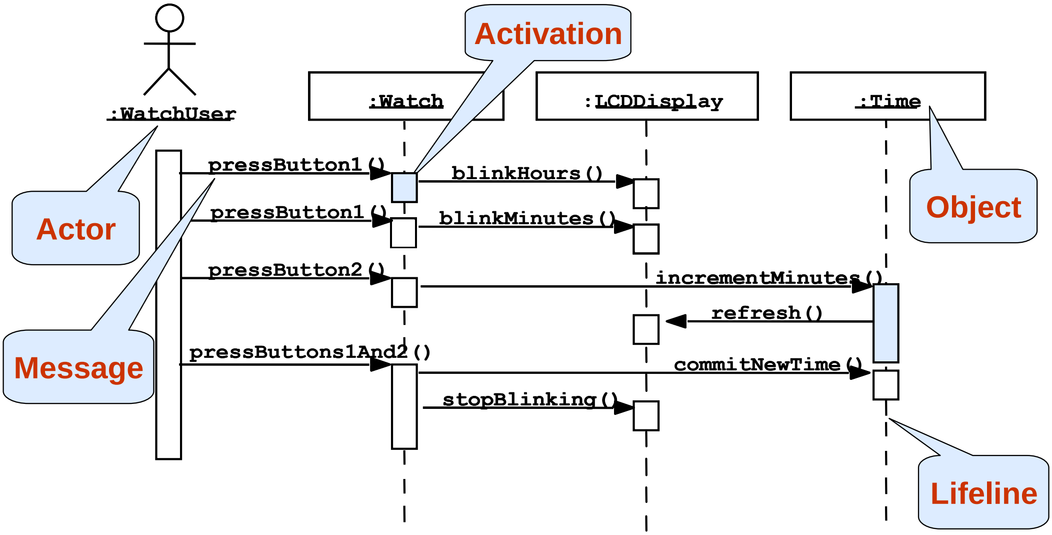
Един use case описва една от функционалностите на системата. Той има уникално име и е свързан с актьори. Може да има входни и изходни условия. Най-често съдържа поток от действия (процес). Може да има и други изисквания.

Sequence диаграми

Използват се при моделиране на изискванията за описание на процеси. За по-добро описание на use case сценариите. Позволяват описание на допълнителни участници в процесите. Използват се при дизайна за описание на системните интерфейси.

Sequence диаграми – пример

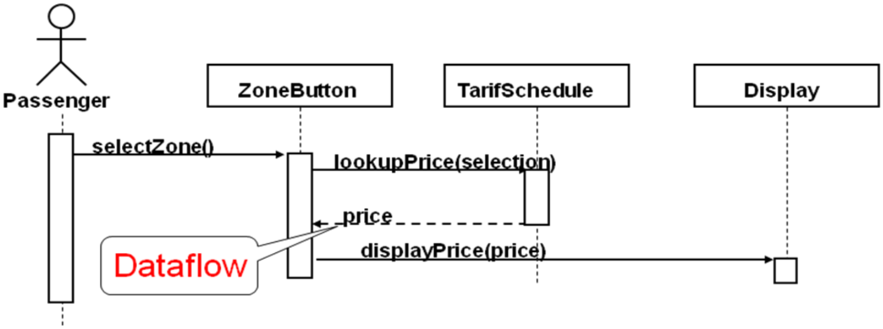
Ето как изглежда една sequence диаграма:


Activation
:WatchUser
:Watch
:LCDDisplay
:Time
Actor
Object
Message
Lifeline
pressButton1()
blinkHours()
pressButton1()
blinkMinutes()
pressButton2()
incrementMinutes()
refresh()
pressButtons1And2()
commitNewTime()
stopBlinking()

Класовете се представят с колони. **Съобщенията (действията)** се представят чрез стрелки. **Участниците** се представят с широки правоъгълници. **Състоянията** се представят с пунктирана линии.

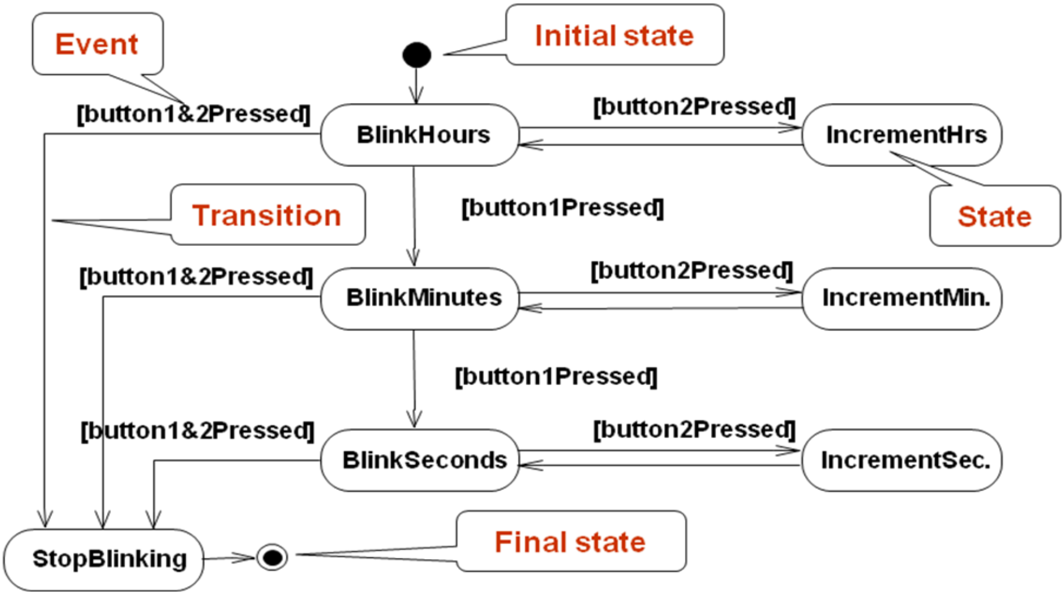
Съобщения – пример

Посоката на стрелката определя изпращача и получателя на съобщението. Хоризонталните прекъснати линии изобразяват потока на данните:


Passenger
ZoneButton
TarifSchedule
Display
selectZone()
lookupPrice(selection)
price
displayPrice(price)
Dataflow

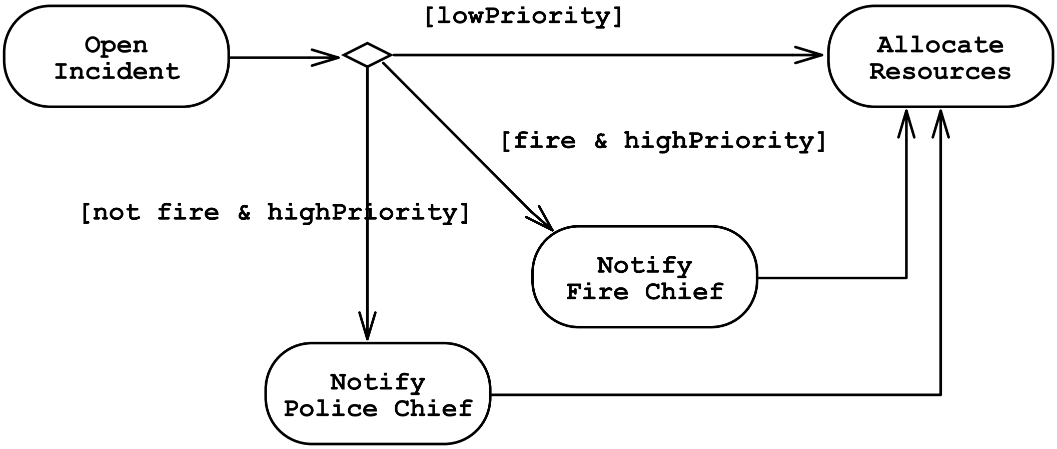
Statechart диаграми

Statechart диаграмите описват възможните състояния на даден процес и възможните преходи между тях. Представляват краен автомат:


Event
Initial state
[button1&2Pressed]
BlinkHours
[button2Pressed]
IncrementHrs
State
Transition
[button1Pressed]
[button1&2Pressed]
BlinkMinutes
[button2Pressed]
IncrementMin.
[button1Pressed]
[button1&2Pressed]
BlinkSeconds
[button2Pressed]
IncrementSec.
StopBlinking
Final state

Activity диаграми

Представляват специален тип statechart диаграми, при които състоянията са действия. Показват потока на действията в системата:


Open Incident
[lowPriority]
Allocate Resources
[fire & highPriority]
[not fire & highPriority]
Notify Fire Chief
Notify Police Chief

Шаблони за дизайн

Достатъчно време след появата на обектно-ориентираната парадигма се оказва, че съществуват множество ситуации, които се появяват често при писането на софтуер. Например клас, който трябва да има само една инстанция в рамките на цялото приложение.

Появяват се **шаблоните за дизайн (design patterns)** – популярни решения на често срещани проблеми от обектно-ориентираното моделиране. Част от тях са най-добре обобщени в едноименната книга на Ерих Гама "Design Patterns: Elements of Reusable Object Oriented Software" (ISBN 0-201-63361-2).

Това е една от малкото книги на компютърна тематика, които остават актуални 15 години след издаването си. Шаблоните за дизайн допълват основните принципи на ООП с допълнителни добре известни решения на добре известни проблеми. Добро място за започване на разучаването им е статията за тях в Уикипедия: [http://en.wikipedia.org/wiki/Design_pattern (computer science)](http://en.wikipedia.org/wiki/Design_pattern_(computer_science)).

Шаблонът Singleton

Това е най-популярният и използван шаблон. Позволява на определен клас да има само една инстанция и дефинира откъде да се вземе тази инстанция. Типични примери са класове, които дефинират връзка към единствени неща (виртуалната машина, операционна система, мениджър на прозорците при графично приложение, файлова система), както и класовете от следващия шаблон (factory).

Шаблонът Singleton – пример

Ето примерна имплементация на шаблона Singleton:

Singleton.cs

```
public class Singleton
{
    // Single instance
    private static Singleton instance;

    // Initialize the single instance
    static Singleton()
    {
        instance = new Singleton();
    }

    // The property for taking the single instance
    public static Singleton Instance
    {
        get
        {
            return instance;
        }
    }

    // Private constructor – protects direct instantialion
    private Singleton() { }
}
```

Имаме скрит конструктор, за да ограничим инстанциите (най-долу). Имаме статична променлива, която държи единствената инстанция. Инициализираме я еднократно в статичния конструктор на класа. Свойството за вземане на инстанцията най-често се казва **`Instance`**.

Шаблонът може да претърпи много оптимизации, например т.нар. "мързеливо инициализиране" (lazy init) на единствената променлива за спестяване на памет, но това е класическата му форма.

Шаблонът Factory Method

Factory method е друг много разпространен шаблон. Той е предназначен да "произвежда" обекти. Инстанцирането на определен обект не се извършва директно, а се прави от factory метода. Това позволява на factory метода да реши коя конкретна инстанция да създаде. Решението може да зависи от външната среда, от параметър или от някаква системна настройка.

Шаблонът Factory Method – пример

Factory методите капсулират създаването на обекти. Това е полезно, ако процесът по създаването е много сложен – например зависи от настройки в конфигурационните файлове или от данни въведени от потребителя.

Нека имаме клас, който съдържа графични файлове (**`png`**, **`jpeg`**, **`bmp`**, …) и създава умалени откъм размер техни копия (т.нар. thumbnails). Поддържат се различни формати представени от клас за всеки от тях:

```
public class Thumbnail
{
    // ...
}

public interface Image
{
    Thumbnail CreateThumbnail();
}

public class GifImage : Image
{
    public Thumbnail CreateThumbnail()
    {
        // ... create thumbnail ...
        return gifThumbnail;
    }
}

public class JpegImage : Image
{
    public Thumbnail CreateThumbnail()
    {
        // ... create thumbnail ...
        return jpegThumbnail;
    }
}
```

Ето го и класът-албум на изображения:

```
public class ImageCollection
{
    private IList<Image> images;

    public ImageCollection(IList<Image> images)
    {
        this.images = images;
    }

    public IList<Thumbnail> CreateThumbnails()
    {
        IList<Thumbnail> thumbnails =
            new List<Thumbnail>(images.Count);

        foreach (Image th in images)
        {
            thumbnails.Add(th.CreateThumbnail());
        }
        return thumbnails;
    }
}
```

Клиентът на програмата може да изисква умалени копия на всички изображения в албума:

```
public class Example
{
    public static void Main()
    {
        IList<Image> images = new List<Image>();

        images.Add(new JpegImage());
        images.Add(new GifImage());

        ImageCollection imageRepository =
            new ImageCollection(images);

        Console.WriteLine(imageRepository.CreateThumbnails());
    }
}
```

Други шаблони

Съществуват десетки други добре известни шаблони за дизайн, но няма да се спираме подробно на тях. По-любознателните читатели могат да потърсят за "Design Patterns" в Интернет и да разберат за какво служат и как се използват шаблони като: Abstract Factory, Prototype, adapter, composite, Façade, Command, Iterator, Observer и много други. Ако продължите да се занимавате с .NET по-сериозно, ще се убедите, че цялата стандартна библиотека (CTS) е конструирана върху принципите на ООП и използва много активно класическите шаблони за дизайн.

Упражнения

1. Нека е дадено едно училище. В училището име класове от ученици. Всеки клас има множество от учители. Всеки учител преподава множество от предмети. Учениците имат име и уникален номер в класа. Класовете имат уникален текстов идентификатор. Учителите имат име. Предметите имат име, брой на часове и брой упражнения. Както учителите, така и студентите са хора. Вашата задача е да моделирате класовете (в контекста на ООП) заедно с техните атрибути и операции, дефинирате класовата йерархия и създайте диаграма с Visual Studio.

2. Дефинирайте клас **`Human`** със свойства "собствено име" и "фамилно име". Дефинирайте клас **`Student`**, наследяващ **`Human`**, който има свойство "оценка". Дефинирайте клас **`Worker`**, наследяващ **`Human`**, със свойства "надница" и "изработени часове". Имплементирайте и метод "изчисли надница за 1 час", който смята колко получава работникът за 1 час работа, на базата на надницата и изработените часове. Напишете съответните конструктори и методи за достъп до полетата (свойства).

3. Инициализирайте масив от 10 студента и ги сортирайте по оценка в нарастващ ред. Използвайте интерфейса **`System.IComparable`**.

4. Инициализирайте масив от 10 работника и ги сортирайте по заплата в намаляващ ред.

5. Дефинирайте клас **`Shape`** със само един метод **`calculateSurface()`** и полета **`width`** и **`height`**. Дефинирайте два нови класа за триъгълник и правоъгълник, които имплементират споменатия виртуален метод. Този метод трябва да връща площта на правоъгълника (**`height*width`**) и триъгълника (**`height*width/2`**). Дефинирайте клас за кръг с подходящ конструктор, при когото при инициализация и двете полета (**`height`** и **`width`**) са с еднаква стойност (радиуса), и имплементирайте виртуалния метод за изчисляване на площта. Направете масив от различни фигури и сметнете площта на всичките в друг масив.

6. Имплементирайте следните обекти: куче (**`Dog`**), жаба (**`Frog`**), котка (**`Cat`**), котенце (**`Kitten`**), котарак (**`Tomcat`**). Всички те са животни (**`Animal`**). Животните се характеризират с възраст (**`age`**), име (**`name`**) и

пол (**`gender`**). Всяко животно издава звук (виртуален метод на **`Animal`**). Направете масив от различни животни и за всяко изписвайте на конзолата името, възрастта и звука, който издава.

7. Изтеглете си някакъв инструмент за работа с UML и негова помощ генерирайте клас диаграма на класовете от предходната задача.
8. Дадена банка предлага различни типове сметки за нейните клиенти: депозитни сметки, сметки за кредит и ипотечни сметки. Клиентите могат да бъдат физически лица или фирми. Всички сметки имат клиент, баланс и месечен лихвен процент. Депозитните сметки дават възможност да се внасят и теглят пари. Сметките за кредит и ипотечните сметки позволяват само да се внасят пари. Всички сметки могат да изчисляват стойността на лихвата си за даден период (в месеци). В общия случай това става като се умножи броят_на_месеците * месечния_лихвен_процент. Кредитните сметки нямат лихва за първите три месеца ако са на физически лица. Ако са на фирми – нямат лихва за първите два месеца. Депозитните сметки нямат лихва ако техният баланс е положителен и по-малък от 1000. Ипотечните сметки имат ½ лихва за първите 12 месеца за фирми и нямат лихва за първите 6 месеца за физически лица. Вашата задача е да напишете обектно-ориентиран модел на банковата система чрез класове и интерфейси. Трябва да моделирате класовете, интерфейсите, базовите класове и абстрактните операции и да имплементирате съответните изчисления за лихвите.
9. Прочетете за шаблона "Abstract Factory" и го имплементирайте.

Решения и упътвания

1. Задачата е тривиална. Просто следвайте условието и напишете кода.
2. Задачата е тривиална. Просто следвайте условието и напишете кода.
3. Имплементирайте **`IComparable`** в **`Student`** и оттам просто сортирайте списъка.
4. Задачата е като предната.
5. Имплементирайте класовете, както са описани в условието на задачата.
6. Изписването на информацията можете да го имплементирате във виртуалния метод **`System.Object.ToString()`**. За да принтирате съдържанието на целия масив, можете да ползвате цикъл с **`foreach`**.
7. Можете да намерите списък с UML инструменти от следния адрес: http://en.wikipedia.org/wiki/List_of_UML_tools.
8. Имплементирайте класовете както са описани в условието на задачата.
9. Можете да прочетете за шаблона "abstract factory" от Wikipedia: http://en.wikipedia.org/wiki/Abstract_factory_pattern.

Глава 21. Качествен програмен код

В тази тема...

В настоящата тема ще разгледаме основните правила за писане на качествен програмен код. Ще бъде обърнато внимание на именуването на елементите от програмата (променливи, методи, класове и други), правилата за форматиране и подреждане на кода, добрите практики за изграждане на висококачествени методи и принципите за качествена документация на кода. Ще бъдат дадени много примери за качествен и некачествен код. Ще бъдат описани и официалните "Design Guidelines for Developing Class Libraries за .NET" от Майкрософт. В процеса на работа ще бъде обяснено как да се използва средата за програмиране, за да се автоматизират някои операции като форматиране и преработка на кода.

Тази тема се базира на предходната – "Принципи на Обектно-ориентираното програмиране" и очаква читателят да е запознат с основните ООП принципи: Абстракция, наследяване, полиморфизъм и капсулация, които имат огромно значение върху качеството на кода.

Защо качеството на кода е важно?

Нека разгледаме следния код:

```
static void Main()
{
  int value=010, i=5, w;
  switch(value){case 10:w=5;Console.WriteLine(w);break;case
9:i=0;break;
          case 8:Console.WriteLine("8 ");break;
     default:Console.WriteLine("def ");{
          Console.WriteLine("hoho "); }
     for (int k = 0; k < i; k++, Console.WriteLine(k -
'f'));break;} { Console.WriteLine("loop!"); }
}
```

Можете ли от първия път да познаете какво прави този код? Дали го прави правилно или има грешки?

Какво е качествен програмен код?

Качеството на една програма има два аспекта – качеството, измерено през призмата на потребителя (наречено **външно качество**), и от гледна точка на вътрешната организация (наречено **вътрешно качество**).

Външното качество зависи от това колко коректно работи тази програма. Зависи също от това колко е интуитивен и ползваем е потребителският интерфейс. Зависи и от производителността (колко бързо се справя тя с поставените задачи).

Вътрешното качество е свързано с това колко добре е построена тази програма. То зависи от архитектурата и дизайна (дали са достатъчно изчистени и подходящи). Зависи от това колко лесно е да се направи промяна или добавяне на нова функционалност (леснота за поддръжка). Зависи и от простотата на реализацията и четимостта на кода. Вътрешното качество е свързано най-вече с кода на програмата.

Характеристики за качество на кода

Качествен програмен код е такъв, който се чете и разбира лесно. Качествен код е такъв, който се модифицира и поддържа лесно и праволинейно. Той трябва да е коректен при всякакви входни данни, да е добре тестван. Трябва да има добра архитектура и дизайн. Документацията трябва да е на ниво или поне кодът да е самодокументиращ се. Трябва да има добро форматиране, което консистентно се прилага навсякъде.

На всички нива (модули, класове, методи) трябва да има висока свързаност на отговорностите (strong cohesion) – едно парче код трябва да върши точно едно определено нещо.

Функционалната независимост (loose coupling) между модули, класове и методи е от изключителна важност. Подходящо и консистентно именуване на класовете, методите, променливите и останалите елементи също е задължително условие. Кодът трябва да има и добра документация, вградена в него самия.

Защо трябва да пишем качествено?

Нека погледнем този код отново:

```
static void Main()
{
  int value=010, i=5, w;
  switch(value){case 10:w=5;Console.WriteLine(w);break;case
9:i=0;break;
          case 8:Console.WriteLine("8 ");break;
     default:Console.WriteLine("def ");{
          Console.WriteLine("hoho "); }
     for (int k = 0; k < i; k++, Console.WriteLine(k -
'f'));break;} { Console.WriteLine("loop!"); }
}
```

Можете ли да кажете дали този код се компилира без грешки? Можете ли да кажете какво прави само като го гледате? Можете ли да добавите нова функционалност и да сте сигурни, че няма да счупите нищо старо? Можете ли да кажете за какво служи променливата **k** или променливата **w**?

Във Visual Studio има опция за пренареждане на код. Ако горният код бъде сложен в Visual Studio и се извика тази опция (клавишна комбинация [Ctrl+K, Ctrl+F]), кодът ще бъде преформатиран и ще изглежда съвсем различно. Въпреки това все още няма да е ясно за какво служат променливите, но поне ще е ясно кой блок с код къде завършва:

```
static void Main()
{
  int value = 010, i = 5, w;
  switch (value)
  {
     case 10: w = 5; Console.WriteLine(w); break;
     case 9: i = 0; break;
     case 8: Console.WriteLine("8 "); break;
     default: Console.WriteLine("def ");
       {
          Console.WriteLine("hoho ");
       }
```

```
        for (int k = 0; k < i; k++, Console.WriteLine(k - 'f')) ;
break;
   } { Console.WriteLine("loop!"); }
}
```

Ако всички пишеха код както в примера, нямаше да е възможно реализирането на големи и сериозни софтуерни проекти, защото те се пишат от големи екипи от софтуерни инженери. Ако кодът на всички е като в примера по-горе, никой няма да е в състояние да разбере как работи (и дали работи) кодът на другите от екипа, а с голяма вероятност никой няма да си разбира и собствения код.

С времето в професията на програмистите се е натрупал сериозен опит и добри практики за писане на качествен програмен код, за да е възможно всеки да разбере кода на колегите си и да може да го променя и дописва. Тези практики представляват множество от препоръки и правила за форматиране на кода, за именуване на идентификаторите и за правилно структуриране на програмата, които правят писането на софтуер по-лесно. Качественият и консистентен код помага най-вече за поддръжката и лесната промяна. Качественият код е гъвкав и стабилен. Той се чете и разбира лесно от всички. Ясно е какво прави от пръв поглед, поради това е самодокументиращ се. Качественият код е интуитивен – ако не го познавате има голяма вероятност да познаете какво прави само с един бърз поглед. Качественият код е удобен за преизползване, защото прави само едно нещо (strong cohesion), но го прави добре, като разчита на минимален брой други компоненти (loose coupling) и ги използва само през публичните им интерфейси. Качественият код спестява време и труд и прави написания софтуер по-ценен.

Някои програмисти гледат на качествения код като на прекалено прост. Не смятат, че могат да покажат знанията си с него. И затова пишат трудно четим код, който използва характеристики, които не са добре документирани или не са популярни. Пишат функции на един ред. Това е изключително грешна практика.

Код-конвенции

Преди да продължим с препоръките за писане на качествен програмен код ще поговорим малко за код-конвенции. **Код-конвенция** е група правила за писане на код, използвана в рамките на даден проект или организация. Те могат да включват правила за именуване, форматиране и логическа подредба. Едно такова правило например може да препоръчва класовете да започват с главна буква, а променливите – с малка. Друго правило може да твърди, че къдравата скоба за нов блок с програмни конструкции се слага на същия ред, а не на нов ред.

Неконсистентното използване на една конвенция е по-лошо и по-опасно от липсата на конвенция въобще.

Конвенциите са започнали да се появяват в големи и сериозни проекти, в които голям брой програмисти са пишели със собствен стил и всеки от тях е спазвал собствени (ако въобще е спазвал някакви) правила. Това е правело кода по-трудно четим и е принудило ръководителите на проектите да въведат писани правила. По-късно най-добрите код конвенции са придобили популярност и са станали де факто стандарт.

Microsoft има официална код конвенция наречена **Design Guidelines for Developing Class Libraries** (http://msdn.microsoft.com/en-us/library/ms229042(VS.100).aspx за .NET Framework 4.0).

От тогава тази код конвенция е добила голяма популярност и е широко разпространена. Правилата за именуване на идентификаторите и за форматиране на кода, които ще дадем в тази тема, са в синхрон с код конвенцията на Microsoft.

Големите организации спазват стриктни конвенции, като конвенциите в отделните екипи могат да варират. Повечето водачи на екипи избират да спазват официалната конвенция на Microsoft като в случаите в които тя не е достатъчна се разширява според нуждите.

Качеството на кода не е група конвенции, които трябва да се спазват, то е начин на мислене.

Управление на сложността

Управлението на сложността играе централна роля в писането на софтуер. Основната цел е да се намали сложността, с която всеки трябва да се справя. Така мозъкът на всеки един от участниците в създаването на софтуер се налага да мисли за много по-малко неща.

Управлението на сложността започва от архитектурата и дизайна. Всеки един от модулите (или автономните единици код) или дори класовете трябва да са проектирани, така че да намаляват сложността.

Добрите практики трябва да се прилагат на всяко ниво – класове, методи, член-променливи, именуване, оператори, управление на грешките, форматиране, коментари. Добрите практики са в основата на намаляване на сложността. Те канализират много решения за кода по строго определени правила и така помагат на всеки един разработчик да мисли за едно нещо по-малко докато чете и пише код.

За управлението на сложността може да се гледа и от частното към общото: за един разработчик е изключително полезно да може да се абстрахира от голямата картина, докато пише едно малко парче код. За да е възможно това, парчето код трябва да е с достатъчно ясни очертания съобразени с голямата картина. Важи римското правило - разделяй и владей, но отнесено към сложността.

Правилата, за които ще говорим по-късно са насочени точно към това, да се намери начин цялостната сложност да бъде "изключена" докато се работи над една малка част от системата.

Именуване на идентификаторите

Идентификатори са имената на класове, интерфейси, изброими типове, анотации, методи и променливи. В C# и в много други езици имената на идентификаторите се избират от разработчика. Имената не трябва да бъдат случайни. Те трябва да са съставени така, че да носят полезна информация за какво служат и каква точно роля изпълняват в съответния код. Така кодът става по-лесно четим.

Когато именуваме идентификатори е добре да си задаваме въпроси: Какво прави този клас? Каква е целта на тази променлива? За какво се използва този метод?

Добри имена са:

```
FactorialCalculator, studentsCount, Math.PI, configFileName, CreateReport
```

Лоши имена са:

```
k, k2, k3, junk, f33, KJJ, button1, variable, temp, tmp,
temp_var, something, someValue
```

Изключително лошо име на клас или метод е **`Problem12`**. Някои начинаещи програмисти дават такова име за решението на задача 12 от упражненията. Това е изключително грешно! Какво ще ви говори името **`Problem12`** след 1 седмица или след 1 месец? Ако задачата търси път в лабиринт, дайте и име **`PathInLabyrinth`**. След 3 месеца може да имате подобна задача и да трябва да намерите задачата за лабиринта. Как ще я намерите, ако не сте й дали подходящо име? Не давайте име, което съдържа числа – това е индикация за лошо именуване.

Името на идентификаторите трябва да описва за какво служи този клас. Решението на задача 12 от упражненията не трябва да се казва `Problem12` **или** `Zad12`**. Това е груба грешка!**

Избягвайте съкращения

Съкращения трябва се избягват, защото могат да бъдат объркващи. Например за какво ви говори името на клас **`GrBxPnl`**? Не е ли по-ясно, ако името е `GroupBoxPanel`? Изключения се правят за акроними, които са по-популярни от пълната си форма, например HTML или URL. Например името `HTMLParser` е препоръчително пред `HyperTextMarkupLanguageParser`.

Английски език

Едно от най-основните правила е, винаги да се използва английски език. Помислете само ако някой виетнамец използва виетнамски език, за да си кръщава променливите и методите. Какво ще разберете, ако четете неговия код? Ами какво ще разбере виетнамецът, ако вие сте ползвали български и след това се наложи той да допише вашия код. Единственият език, който всички програмисти владеят, е английският.

⚠	**Английският език е де факто стандарт при писането на софтуер. Винаги използвайте английски език за имената на идентификаторите в сорс кода (променливи, методи, класове и т.н.). Използвайте английски и за коментарите в програмата.**

Нека сега разгледаме как да подберем подходящите идентификатори в различните случаи.

Последователност при именуването

Начинът на именуване трябва да е последователен.

В групата методи `LoadFile()`, `LoadImageFromFile()`, `LoadSettings()`, `LoadFont()`, `LoadLibrary()` е неправилно да се включи и `ReadTextFile()`.

Противоположните дейности трябва да симетрично именувани (тоест, когато знаете как е именувана една дейност, да можете да предположите как е именувана противоположната дейност): `LoadLibrary()` и `UnloadLibrary()`, но не и `FreeHandle()`. Също и `OpenFile()` с `CloseFile()`, но не и `DeallocateResource()`. Към двойката `GetName`, `SetName` е неестествено да се добави `AssignName`.

Забележете, че в CTS големи групи класове имат последователно именуване: колекциите (пакетът и всички класове използват думите Collection и List и никога не използват техни синоними), потоците винаги са Streams.

⚠	**Именувайте последователно – не използвайте синоними. Именувайте противоположностите симетрично.**

Имена на класове, интерфейси и други типове

От главата "Принципи на обектно-ориентираното програмиране" знаем, че класовете описват обекти от реалния свят. Имената на класовете трябва да са съставени от съществително име (нарицателно или собствено), като може да има едно или няколко прилагателни (преди или след съществителното). Например класът описващ Африканския лъв ще се казва **`AfricanLion`**. Тази нотация на именуване се нарича **Pascal Case** – първата буква на всяка дума от името е главна, а останалите са малки. Така по-лесно се чете (за да се убедите в това, забележете разликата между името **`idatagridcolumnstyleeditingnotificationservice`** срещу името **`IDataGridColumnStyleEditingNotificationService`**). Последното име е на публичния клас с най-дълго име в .NET Framework (46 знака, от **`System.Windows.Forms`**).

Да дадем още няколко примера. Трябва да напишем клас, който намира прости числа в даден интервал. Добро име за този клас е **`PrimeNumbers`** или **`PrimeNumbersFinder`** или **`PrimeNumbersScanner`**. Лоши имена биха могли да бъдат **`FindPrimeNumber`** (не трябва да ползваме глагол за име на клас) или **`Numbers`** (не става ясно какви числа и какво ги правим) или Prime (не трябва името на клас да е прилагателно).

Колко да са дълги имената на класовете?

Имената на класовете не трябва да надвишават в общия случай 20 символа, но понякога това правило не се спазва, защото се налага да се опише обект от реалността, който се състои от няколко дълги думи. Както видяхме по-горе има класове и с по 46 знака. Въпреки дължината е ясно за какво този клас. По тази причината препоръката за дължина до 20 символа, е само ориентировъчна, а не задължителна. Ако може едно име да е по-кратко и също толкова ясно, колкото дадено по-дълго име, предпочитайте по-краткото.

Лош съвет би бил да се съкращава, за да се поддържат имената кратки. Следните имена достатъчно ясни ли са: **`CustSuppNotifSrvc`**, **`FNFException`**? Очевидно не са. Доста по-ясни са **`FileNotFoundException`**, **`CustomerSupportNotificationService`**, въпреки че са по-дълги.

Имена на интерфейси и други типове

Имената на интерфейсите трябва да следват същата конвенция, както имената на класовете: изписват се в Pascal Case и се състоят от съществително и евентуално прилагателни. За да се различават от останалите типове, конвенцията повелява да се сложи префикс I.

Примери са **`IEnumerable`**, **`IFormattable`**, **`IDataReader`**, **`IList`**, **`IHttpModule`**, **`ICommandExecutor`**.

Лоши примери са: **`List`**, **`FindUsers`**, **`IFast`**, **`IMemoryOptimize`**, **`Optimizer`**, **`FastFindInDatabase`**, **`CheckBox`**.

В .NET има още една нотация за имена интерфейси: да завършват на "**`able`**": **`Runnable`**, **`Serializable`**, **`Cloneable`**. Това са интерфейси, които най-често добавят допълнителна роля към основната роля на един обект. Повечето интерфейси обаче не следват тази нотация, например интерфейсите **`IList`** и **`ICollection`**.

Имена на изброимите типове (enumerations)

Няколко формата са допустими: **`[Съществително]`** или **`[Глагол]`** или **`[Прилагателно]`**. Имената им са в единствено или множествено число. За всички членове на изброимите типове трябва да се спазва един и същ стил.

```
enum Days
{
  Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
};
```

Имена на атрибути

Имената на атрибутите трябва да имат окончание **`Attribute`**. Например **`WebServiceAttribute`**.

Имена на изключения

Код конвенцията повелява изключенията да завършват на **`Exception`**. Името трябва да е достатъчно информативно. Добър пример би бил **`FileNotFoundException`**. Лош би бил **`FileNotFoundError`**.

Имена на делегати

Делегатите трябва да имат суфикс **`Delegate`** или **`EventHandler`**. **`DownloadFinishedDelegate`** би бил добър пример, докато **`WakeUpNotification`** не би спазвал конвенцията.

Имена на пакети

Пакетите (namespaces, обяснени в главата "Създаване и използване на обекти") използват PascalCase за именуване, също като класовете. Следните формати са за предпочитане: **`Company.Product.Component...`** и **`Product.Component...`** .

Добър пример: **`Telerik.WinControls.GridView`**.

Лош пример: **`Telerik_WinControlsGridView, Classes`**.

Имена на асемблита

Имената на асемблитата съвпадат с името на основния пакет. Добри примери са:

- **`Telerik.WinControls.GridView.dll`**

- **`Oracle.DataAccess.dll`**
- **`Interop.CAPICOM.dll`**

Неправилни имена:

- **`Telerik_WinControlsGridView.dll`**
- **`OracleDataAccess.dll`**

Имена на методи

В имената на методите отново всяка отделна дума трябва да е с главна буква – **PascalCase**.

Имената на методите трябва да се съставят по схемата <глагол> + <обект>, например **`PrintReport()`**, **`LoadSettings()`** или **`SetUserName()`**. Обектът може да е съществително или да е съставен от съществително и прилагателно, например **`ShowAnswer()`**, **`ConnectToRandomTorrentServer()`** или **`FindMaxValue()`**.

Името на метода трябва да отговаря на въпроса какво извършва метода. Ако не можете да измислите добро име, вероятно трябва да преразгледате самия метод и дали е удачно написан.

Като примери за лоши имена на методи можем да дадем следните: **`DoWork()`** (не става ясно каква точно работа върши), **`Printer()`** (няма глагол), **`Find2()`** (ами защо не е **`Find7()`**?), **`ChkErr()`** (не се препоръчват съкращения), **`NextPosition()`** (няма глагол).

Понякога единични глаголи са също добро име за метод, стига да става ясно какво прави съответния метод и върху какви обекти оперира. Например ако имаме клас **`Task`**, методите **`Start()`**, **`Stop()`** и **`Cancel()`** са с добри имена, защото става ясно, че стартират, спират или оттеглят изпълнението на задачата, в текущия обект (**`this`**). В други случаи единичния глагол е грешно име, примерно в клас с име **`Utils`** методи с имена **`Evaluate()`**, **`Create()`** или **`Stop()`** са неадекватни, защото няма контекст.

Методи, които връщат стойност

Имената на методите, които връщат стойност, трябва да описват връщаната стойност, например **`GetNumberOfProcessors()`**, **`FindMinPath()`**, **`GetPrice()`**, **`GetRowsCount()`**, **`CreateNewInstance()`**.

Примери за лоши имена на методи, които връщат стойност (функции) са следните: **`ShowReport()`** (не става ясно какво връща методът), **`Value()`** (трябва да е **`GetValue()`** или **`HasValue()`**), **`Student()`** (няма глагол), **`Empty()`** (трябва да е **`IsEmpty()`**).

Когато се връща стойност трябва да я ясна мерната единица: `MeasureFontInPixels(...)`, а не `MeasureFont(...)`.

Единствена цел на методите

Един метод, който извършва няколко неща е трудно да бъде именуван – какво име ще дадете на метод, който прави годишен отчет на приходите, сваля обновления на софтуера от интернет и сканира системата за вируси? Например `CreateAnnualIncomesReportDownloadUpdatesAndScanForViruses`?

Методите трябва да имат една единствена цел, т.е. да решават само една задача, не няколко едновременно!

Методи с няколко цели (weak cohesion) не могат и не трябва да се именуват правилно. Те трябва да се преработят.

Свързаност на отговорностите и именуване

Името трябва да описва всичко, което методът извършва. Ако не може да се намери подходящо име, значи няма силна свързаност на отговорностите (strong cohesion), т.е. методът върши много неща едновременно и трябва да се раздели на няколко отделни метода.

Ето един пример: имаме метод, който праща e-mail, печата отчет на принтер и изчислява разстояние между точки в тримерното евклидово пространство. Какво име ще му дадем? Може би ще го кръстим `SendEmailAndPrintReportAndCalc3DDistance()`? Очевидно е, че нещо не е наред с този метод – трябва да преработим кода вместо да се мъчим да дадем добро име. Още по-лошо е, ако дадем грешно име, примерно `SendEmail()`. Така подвеждаме всички останали програмисти, че този метод праща поща, а той всъщност прави много други неща.

Даването на заблуждаващо име за метод е по-лошо дори от това да го кръстим `method1()`. Например ако един метод изчислява косинус, а ние му дадем за име `sqrt()`, ще си навлечем яростта на всички колеги, които се опитват да ползват нашия код.

Колко да са дълги имената на методите?

Тук важат същите препоръки като за класовете – не трябва да се съкращава, ако не е ясно.

Добри примери са имената: `LoadCustomerSupportNotificationService()`, `CreateMonthlyAndAnnualIncomesReport()`.

Лоши примери са `LoadCustSuppSrvc()`, `CreateMonthIncReport()`.

Параметри на методите

Параметрите имат следния вид: [Съществително] или [Прилагателно]+[Съществително]. Всяка дума от името трябва да е с главна буква, с изключение на първата, тази нотация се нарича **camelCase**. Както и при всеки друг елемент от кода и тук именуването трябва да е смислено и да носи полезна информация.

Добри примери: `firstName`, `report`, `usersList`, `fontSizeInPixels`, `speedKmH`, `font`.

Лоши примери: `p`, `p1`, `p2`, `populate`, `LastName`, `last_name`, `convertImage`.

Имена на свойства

Имената на свойствата са нещо средно между имената на методите и на променливите – започват с главна буква (PascalCase), но нямат глагол (като променливите). Името им се състои от (прилагателно+) съществително.

Ако имаме свойство `X` е недобра практика да имаме и метод `GetX()` – ще бъде объркващо.

Ако свойството е енумерация, можете да се замислите дали да не кръстите свойството на самата енумерация. Например ако имаме енумерация с име `CacheLevel`, то и свойството може да се кръсти `CacheLevel`.

Имена на променливи

Имената на променливите (променливи използвани в метод) и член-променливите (променливи използвани в клас) според Microsoft конвенцията трябва да спазват camelCase нотацията.

Променливите трябва да имат добро име като всички други елементи на кода. Добро име е такова, което ясно и точно описва обекта, който променливата съдържа. Например добри имена на променливи са `account`, `blockSize` и `customerDiscount`. Лоши имена са: `r18pq`, `__hip`, `rcfd`, `val1`, `val2`.

Името трябва да адресира проблема, който решава променливата. Тя трябва да отговаря на въпроса "какво", а не "как". В този смисъл добри имена са `employeeSalary`, `employees`. Лоши имена са, несвързаните с решавания проблем имена `myArray`, `customerFile`, `customerHashTable`.

Предпочитайте имена от бизнес домейна, в който ще оперира софтуера – `CompanyNames` срещу `StringArray`.

Оптималната дължина на името на променлива е от 10 до 16 символа. Изборът на дължината на името зависи от обхвата – променливите с по-голям обхват и по-дълъг живот имат по-дълго и описателно име:

```
protected Account[] customerAccounts;
```

Променливите с малък обхват и кратък живот могат да са по-кратки:

```
for (int i=0; i < customers.Length; i++) { … }
```

Имената на променливите трябва да са разбираеми без предварителна подготовка. Поради тази причина не е добра идея да се премахват гласните от името на променливата с цел съкращение – **btnDfltSvRzlts** не е много разбираемо име.

Най-важното е, че каквито и правила да бъдат изградени за именуване на променливите, те трябва да бъдат консистентно прилагани навсякъде из кода, в рамките на всички модули на целия проект и от всички членове на екипа. Неконсистентно прилаганото правило е по-опасно от липсата на правило въобще.

Имена на булеви елементи

Параметрите, свойствата и променливите могат да бъдат от булев тип. В тази точка ще опишем спецификата на този тип елементи.

Имената им трябва да дават предпоставка за истина или лъжа. Например: **canRead**, **available**, **isOpen**, **valid**. Примери за неадекватни имена на булеви променливи са: **student**, **read**, **reader**.

Би било полезно булевите елементи да започват с **is**, **has** или **can** (с големи букви за свойствата), но само ако това добавя яснота.

Не трябва да се използват отрицания (предполагат префикса not), защото се получават следните странности:

```
if (! notFound) { … }
```

Добри примери: **hasPendingPayment**, **customerFound**, **validAddress**, **positiveBalance**, **isPrime**.

Лоши примери: **notFound**, **run**, **programStop**, **player**, **list**, **findCustomerById**, **isUnsuccessfull**.

Имена на константи

В C# константите са статични непроменими променливи и се дефинират по следния начин:

```
public struct Int32
{
  public const int MaxValue = 2147483647;
}
```

Имената на константите трябва да се изписват изцяло с главни букви с долна черта между думите. Пример:

```
public static class Math
{
  public const double PI = 3.14159;
}
```

Имената на константите точно и ясно трябва да описват смисъла на даденото число, стринг или друга стойност, а не самата стойност. Например, ако една константа се казва **number314159**, тя е безполезна.

Именуване на специфични типове данни

Имената на променливи, използвани за броячи, е хубаво да включват в името си дума, която указва това, например **usersCount**, **rolesCount**, **filesCount**.

Променливи, които се използват за описване на състояние на даден обект, трябва да бъдат именувани подходящо. Ето няколко примера: **threadState**, **transactionState**.

Временните променливи най-често са с безлични имена (което указва, че са временни променливи, т.е. имат много кратък живот). Добри примери са **index**, **value**, **count**. Неподходящи имена са **a**, **aa**, **tmpvar1**, **tmpvar2**.

Именуване с префикси или суфикси

В по-старите езици (например C) съществуват префиксни или суфиксни нотации за именуване. Много популярна в продължение на много години е била Унгарската нотация. Унгарската нотация е префиксна конвенция за именуване, чрез която всяка променлива получава префикс, който обозначава типа ѝ или предназначението ѝ. Например в Win32 API името **lpcstrUserName** би означавало променлива, която представлява указател към масив от символи, който завършва с 0 и се интерпретира като стринг.

В .NET подобни конвенции не са придобили популярност, защото средите за разработка показват типа на всяка променлива. Изключение донякъде правят графични библиотеки.

Форматиране на кода

Форматирането, заедно с именуването, е едно от основните изисквания за четим код. Без форматиране, каквито и правила да спазваме за имената и структурирането на кода, кодът няма да се чете лесно.

Целите на форматирането са две – по-лесно четене на кода и (следствието от първата цел) по-лесно поддържане на кода. Ако форматирането прави кода по-труден за четене, значи не е добро. Всяко форматиране (отместване, празни редове, подреждане, подравняване и т.н.) може да донесе както ползи, така и вреди. Важно е форматирането на кода да следва логическата структура на програмата, така че да подпомага четенето и логическото й разбиране.

Форматирането на програмата трябва да разкрива нейната логическа структура. Всички правила за форматиране на кода имат една и съща цел – подобряване на четимостта на кода чрез разкриване на логическата му структура.

В средите за разработка на Microsoft кодът може да се форматира автоматично с клавишната комбинация [Ctrl+K, Ctrl+F]. Могат да бъдат зададени различни стандарти за форматиране на код – Microsoft конвенцията, както и потребителски дефинирани стандарти.

Сега ще разгледаме правилата за форматиране от код-конвенцията на Microsoft за C#.

Защо кодът има нужда от форматиране?

```
public   const    string                    FILE_NAME
="example.bin"  ;  static void Main    (            ){
FileStream   fs=     new FileStream(FILE_NAME,FileMode
.   CreateNew)   // Create the writer      for data  .
;BinaryWriter w=new BinaryWriter      (   fs      );//
Write data to                               Test.data.
for(  int i=0;i<11;i++){w.Write((int)i);}w   .Close();
fs   .   Close  (  ) // Create the reader    for data.
;fs=new FileStream(FILE_NAME,FileMode.            Open
,  FileAccess.Read)     ;BinaryReader                r
= new BinaryReader(fs);  // Read data from  Test.data.
 for (int i = 0; i < 11; i++){ Console      .WriteLine
(r.ReadInt32                                       ())
;}r       .    Close    (  );  fs .  Close   ( )  ;  }
```

Може би този код е достатъчен като отговор?

Форматиране на блокове

Блоковете се заграждат с { и }. Те трябва да са на отделни редове. Съдържанието на блока трябва да е изместено навътре с една табулация:

```
if ( some condition )
{
  // Block contents indented by a single [Tab]
  // Don't use spaces for indentation
}
```

Това правило важи за пакети, класове, методи, условни конструкции, цикли и т.н.

Вложените блокове се отместват допълнително. Тук тялото на класа е отместено от тялото на пакета, тялото на метода е отместено допълнително, както и съдържанието на условната конструкция:

```
namespace Chapter_21_Quality_Code
{
   public class IndentationExample
   {
      private int Zero()
      {
         if (true)
         {
            return 0;
         }
      }
   }
}
```

Правила за форматиране на метод

Съгласно конвенцията за писане на код, препоръчана от Microsoft, е добре да се спазват някои правила за форматиране на кода, при декларирането на методи.

Форматиране на множество декларации на методи

Когато в един клас имаме повече от един метод, трябва да разделяме декларациите им с един празен ред:

IndentationExample.cs

```
public class IndentationExample
{

   public static void DoSth1()
   {
      // ...
   }// Follows one blank line

   public static void DoSth2()
   {
      // ...
   }
}
```

Как да поставяме кръгли скоби?

В конвенцията за писане на код, на Microsoft, се препоръчва, между ключова дума, като например – **`for`**, **`while`**, **`if`**, **`switch`**… и отваряща скоба да поставяме интервал:

```
while (!EOF)
{
   // ... Code ...
}
```

Това се прави с цел да се различават по-лесно ключовите думи.

При имената на методите не се оставя празно място преди отварящата кръгла скоба.

```
public void CalculateCircumference(int radius)
{
   return 2 * Math.PI * radius;
}
```

В този ред на мисли, между името на метода и отварящата кръгла скоба – "(", **не трябва** да има невидими символи (интервал, табулация и т.н.):

```
public static void PrintLogo()
{
   // ... Code ...
}
```

Форматиране на списъка с параметри на методи

Когато имаме метод с много параметри, трябва добре да оставяме един интервал разстояние между поредната запетайка и типа на следващия параметър, но не и преди запетаята:

```
public void CalcDistance(Point startPoint, Point endPoint)
```

Съответно, същото правило прилагаме, когато извикваме метод с повече от един параметър. Преди аргументите, предшествани от запетайка, поставяме интервал:

```
DoSmth(1, 2, 3);
```

Правила за форматирането на типове

Когато създаваме класове, интерфейси, структури или енумерации също е добре да следваме няколко препоръки от Microsoft за форматиране на кода в класовете.

Правила за подредбата на съдържанието на класа

Както знаем, на първия ред се декларира името на класа, предхождано от ключовата дума **`class`**:

```
public class Dog
{
```

След това се декларират константите, като първо се декларират тези с модификатор за достъп **`public`**, след това тези с **`protected`** и накрая – с **`private`**:

```
   // Static variables
   public const string SPECIES = "Canis Lupus Familiaris";
```

След тях се декларират и нестатичните полета. По подобие на статичните, първо се декларират тези с модификатор за достъп **`public`**, след това тези с **`protected`** и накрая – тези с **`private`**:

```
   // Instance variables
   private int age;
```

След нестатичните полета на класа, идва ред на декларацията на конструкторите:

```
   // Constructors
   public Dog(string name, int age)
   {
      this.Name = name;
      this.age = age;
   }
```

След конструкторите се декларират свойствата:

```
   // Properties
   public string Name { get; set; }
```

Най-накрая, след свойствата, се декларират методите на класа. Препоръчва се да групираме методите по функционалност, вместо по ниво на достъп или област на действие. Например, метод с модификатор за достъп **`private`**, може да бъде между два метода с модификатори за достъп – **`public`**. Целта на всичко това е да се улесни четенето и разбирането на кода. Завършваме със скоба за край на класа:

```
   // Methods
   public void Breath()
   {
    // TODO: breathing process
  }
  public void Bark()
  {
    Console.WriteLine("wow-wow");
  }
}
```

Правила за форматирането на цикли и условни конструкции

Форматирането на цикли и условни конструкции става по правилата за форматиране на методи и класове. Тялото на условна конструкция или цикъл задължително се поставя в блок, започващ с "{" и завършващ със "}". Първата скоба се поставя на нов ред, веднага след условието на цикъла или условната конструкция. Тялото на цикъл или условна конструкция задължително се отмества надясно с една табулация. Ако условието е дълго и не се събира на един ред, се пренася на нов ред с две табулации надясно. Ето пример за коректно форматирани цикъл и условна конструкция:

```
public static void Main()
{
  Dictionary<int, string> bulgarianNumbersHashtable =
    new Dictionary<int, string>();
  bulgarianNumbersHashtable.Add(1, "едно");
  bulgarianNumbersHashtable.Add(2, "две");
  bulgarianNumbersHashtable.Add(3, "три");

  foreach (KeyValuePair<int, string> pair in
    bulgarianNumbersHashtable.ToArray())
  {
    Console.WriteLine("Pair: [{0},{1}]", pair.Key, pair.Value);
  }
}
```

Изключително грешно е да се използва отместване от края на условието на цикъла или условната конструкция като в този пример:

```
foreach (Student s in students) {
          Console.WriteLine(s.Name);
          Console.WriteLine(s.Age);
         }
```

Използване на празни редове

Типично за начинаещите програмисти е да поставят безразборно в програмата си празни редове. Наистина, празните редове не пречат, защо да не ги поставяме, където си искаме и защо да ги чистим, ако няма нужда от тях? Причината е много проста: празните редове се използват за разделяне на части от програмата, които не са логическо свързани – празните редове са като начало и край на параграф. Празни редове се поставят за разделяне на методите един от друг, за отделяне на група член-променливи от друга група член-променливи, които имат друга логическа задача, за отделяне на група програмни конструкции от друга група програмни конструкции, които представляват две отделни части на програмата.

Ето един пример с два метода, в който празните редове не са използвани правилно и това затруднява четимостта на кода:

```
public static void PrintList(IList<int> list)
{
  Console.Write("{ ");
  foreach (int item in list)
  {
    Console.Write(item);

    Console.Write(" ");


  }
  Console.WriteLine("}");
}
public static void Main()
{
  IList<int> firstList = new List<int>();
  firstList.Add(1);

  firstList.Add(2);
  firstList.Add(3);
  firstList.Add(4);
  firstList.Add(5);
  Console.Write("firstList = ");
  PrintList(firstList);
  List<int> secondList = new List<int>();
  secondList.Add(2);

  secondList.Add(4);
  secondList.Add(6);
  Console.Write("secondList = ");
  PrintList(secondList);
  List<int> unionList = new List<int>();
```

```
   unionList.AddRange(firstList);
   Console.Write("union = ");

   PrintList(unionList);
}
```

Сами виждате, че празните редове не показват логическата структура на програмата, с което нарушават основното правило за форматиране на кода. Ако преработим програмата, така че да използваме правилно празните редове за отделяне на логически самостоятелните части една от друга, ще получим много по-лесно четим код:

```
public static void PrintList(IList<int> list)
{
   Console.Write("{ ");
   foreach (int item in list)
   {
      Console.Write(item);
      Console.Write(" ");
   }
   Console.WriteLine("}");
}

public static void Main()
{
   IList<int> firstList = new List<int>();
   firstList.Add(1);
   firstList.Add(2);
   firstList.Add(3);
   firstList.Add(4);
   firstList.Add(5);
   Console.Write("firstList = ");
   PrintList(firstList);

   List<int> secondList = new List<int>();
   secondList.Add(2);
   secondList.Add(4);
   secondList.Add(6);
   Console.Write("secondList = ");
   PrintList(secondList);

   List<int> unionList = new List<int>();
   unionList.AddRange(firstList);
   Console.Write("union = ");
   PrintList(unionList);
}
```

Правила за пренасяне и подравняване

Когато даден ред е дълъг, разделете го на два или повече реда, като редовете след първия отместете надясно с една табулация:

```
Dictionary<int, string> bulgarianNumbersHashtable =
   new Dictionary<int, string>();
```

Грешно е да подравнявате сходни конструкции спрямо най-дългата от тях, тъй като това затруднява поддръжката на кода:

```
DateTime          date      = DateTime.Now.Date;
int               count     = 0;
Student           student   = new Strudent();
List<Student>     students  = new List<Student>();
```

Или

```
matrix[x, y]                      == 0;
matrix[x + 1, y + 1]              == 0;
matrix[2 * x + y, 2 * y + x]      == 0;
matrix[x * y, x * y]              == 0;
```

Грешно е да подравнявате параметрите при извикване на метод вдясно спрямо скобата за извикване:

```
Console.WriteLine("word '{0}' is seen {1} times in the text",
                  wordEntry.Key,
                  wordEntry.Value);
```

Същият код може да се форматира правилно по следния начин (този начин не е единственият правилен):

```
Console.WriteLine(
   "word '{0}' is seen {1} times in the text",
   wordEntry.Key,
   wordEntry.Value);
```

Висококачествени класове

Софтуерен дизайн

Когато се проектира една система, често отделните подзадачи се отделят в отделни модули или подсистеми. Задачите, които решават, трябва да са ясно дефинирани. Взаимовръзките между отделните модули също трябва да са ясни предварително, а не да се измислят в движение.

В предишната глава, в която разяснихме ООП, показахме как се използва обектно-ориентираното моделиране за дефиниране на класове от реалните актьори в домейна на решаваната задача. Там споменахме и употребата на шаблони за дизайн.

Добрият софтуерен дизайн е с минимална сложност и е лесен за разбиране. Поддържа се лесно и промените се правят праволинейно (вижте спагети кода в предходната глава). Всяка една единица (метод, клас, модул) е логически свързана вътрешно (strong cohesion), функционално независима и минимално обвързана с други модули (loose coupling). Добре проектираният код се преизползва лесно.

ООП

При създаването на качествени класове основните правила произтичат от четирите принципа на ООП:

Абстракция

Няколко основни правила:

- Едно и също ниво на абстракция при публични членове на класа.
- Интерфейсът на класа трябва да е изчистен и ясен.
- Класът описва само едно нещо.
- Класът трябва да скрива вътрешната си имплементация.

Кодът се развива във времето. Важно е въпреки еволюцията на класовете, техните интерфейси да не се развалят, например:

```
class Employee
{
  public string firstName;
  public string lastName;
  ...
  public SqlCommand FindByPrimaryKeySqlCommand(int id);
}
```

Последният метод е несъвместим с нивото на абстракция, на което работи **Employee**. Потребителят на класа не трябва да знае въобще, че той работи с база от данни вътрешно.

Наследяване

Не скривайте методи в класовете наследници:

```
public class Timer
{
   public void Start() {...}
}
```

```
public class AtomTimer : Timer
{
  public void Start() {...}
}
```

Методът в класа-наследник скрива реалната имплементация. Това не е препоръчително. Ако все пак това поведение е желано (в редките случаи, в които това се налага), се използва ключовата дума **`new`**.

Преместете общи методи, данни, поведение колкото се може по-нагоре в дървото на наследяване. Така тази функционалност няма да се дублира и ще бъде достъпна от по-голяма аудитория.

Ако имате клас, който има само един наследник, смятайте това за съмнително. Това ниво на абстракция може би е излишно. Съмнителен би бил и метод, който пренаписва такъв от базовия клас, който обаче не прави нищо повече от базовия метод.

Дълбокото наследяване с повече от 6 нива е трудно за проследяване и поддържане, затова не е препоръчително. В наследен клас достъпвайте член-променливите през свойства, а не директно.

Следният пример демонстрира кога трябва се предпочете наследяване пред проверка на типовете:

```
switch (shape.Type)
{
  case Shape.Circle:
    shape.DrawCircle();
    break;
  case Shape.Square:
    shape.DrawSquare();
    break;
  ...
}
```

Тук подходящо би било **`Shape`** да бъде наследено от **`Circle`** и **`Square`**, които да имплементират виртуалния метод **`Shape.Draw()`**.

Капсулация

Добър подход е всички членове да бъдат първо **`private`**. Само тези, които е нужно да се виждат навън, се променят първо на **`protected`** и после на **`public`**.

Имплементационните детайли трябва да са скрити. Ползвателите на един качествен клас, не трябва да знаят как той работи вътрешно, за тях трябва да е ясно какво прави той и как се използва.

Член-променливите трябва да са скрити зад свойства. Публичните член-променливи са проява на некачествен код. Константите са изключение.

Публичните членове на един клас трябва да са последователни спрямо абстракцията, която представя този клас. Не правете предположения как ще се използва един клас.

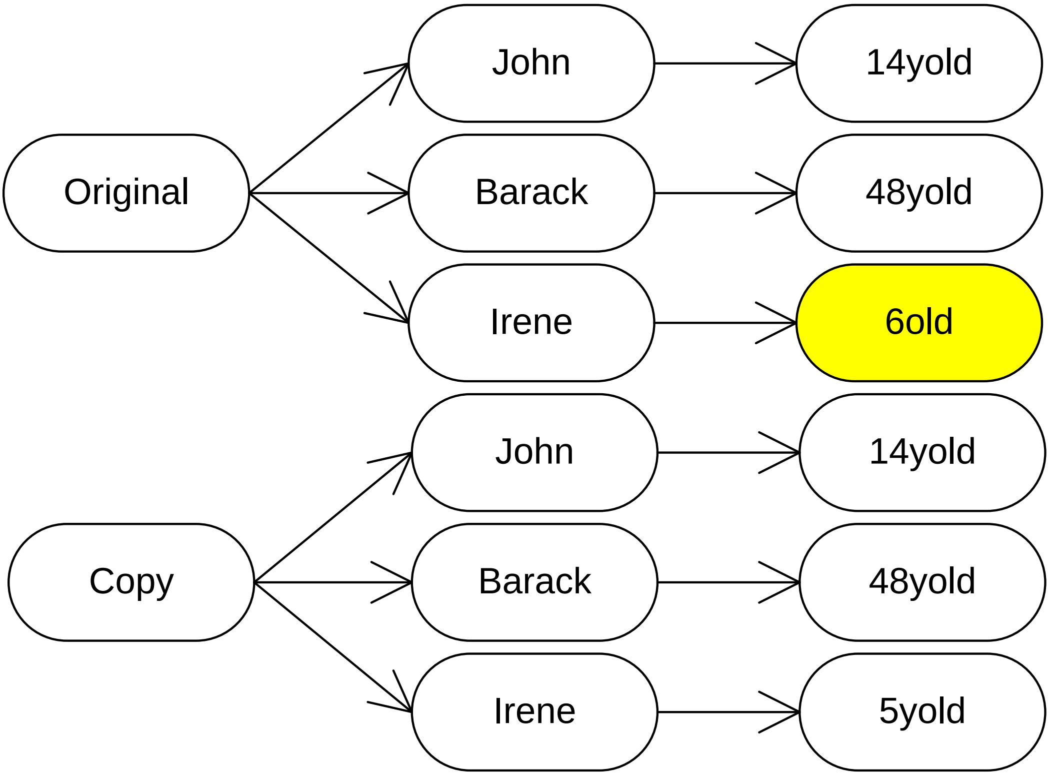
Не разчитайте на недокументирана вътрешна имплементационна логика.

Конструктори

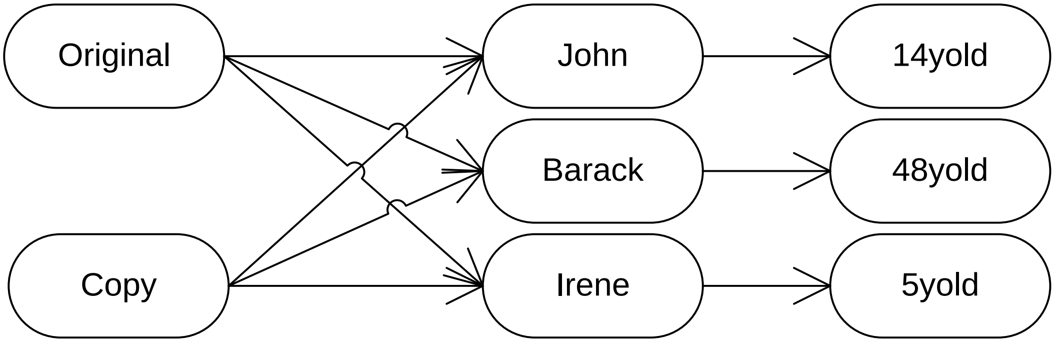
За предпочитане е всички членове на класа да са инициализирани в конструктора. Опасно е използването на неинициализиран клас. Полуинициализиран клас е още по-опасно. Инициализирайте член-променливите в реда, в който са декларирани.

Дълбоко копие на един клас е копие, в което всички член-променливи се копират, и техните член-променливи също се копират. Плитко копие е такова, в което се копират само членовете на първо ниво.

Дълбоко копие:



Плитко копие:



Плитките копия са опасни, защото промяната в един обект води до скрити промени в други. Забележете как във втория пример промяната на възрастта на Ирен в оригинала не води до промяна на възрастта на Ирен в копието. При плитките копия промяната ще се отрази и на двете места.

Висококачествени методи

Качеството на нашите методи е от съществено значение за създаването на висококачествен софтуер и неговата поддръжка. Те правят програмите ни по-четливи и по-разбираеми. Методите ни помагат да намалим сложността на софтуера, да го направим по-гъвкав и по-лесен за модифициране.

От нас зависи, до каква степен ще се възползваме от тези предимства. Колкото по-високо е качеството на методите ни, толкова повече печелим от тяхната употреба. В следващите параграфи ще се запознаем с някои от основните принципи за създаване на качествени методи.

Защо да използваме методи?

Преди да започнем да говорим за добрите имена на методите, нека отделим известно време и да обобщим причините, поради които използваме методи.

Методът решава по-малък проблем. Много методи решават много малки проблеми. Събрани заедно, те решават по-голям проблем – това е римското правило "разделяй и владей" – по-малките проблеми се решават по-лесно.

Чрез методите се намалява сложността на задачата – сложните проблеми се разбиват на по-прости, добавя се допълнително ниво на абстракция, скриват се детайли за имплементацията и се намалява рискът от неуспех. С помощта на методите се избягва повторението на еднакъв код. Скриват се сложни последователности от действия.

Най-голямото предимство на методите е възможността за преизползване на код – те са най-малката преизползваема единица код. Всъщност точно така са възникнали методите.

Какво трябва да прави един метод?

Един метод трябва да върши работата, която е описана в името му и нищо повече. Ако един метод не върши това, което предполага името му, то или името му е грешно, или методът върши много неща едновременно, или просто методът е реализиран некоректно. И в трите случая методът не отговаря на изискванията за качествен програмен код и има нужда от преработка.

Един метод или трябва да свърши работата, която се очаква от него, или трябва да съобщи за грешка. В .NET съобщаването за грешки се осъществява с хвърляне на изключение. При грешни входни данни е недопустимо даден метод да връща грешен резултат. Методът или трябва да работи коректно или да съобщи, че не може да свърши работата си, защото не са на лице необходимите му условия (при некоректни параметри, неочаквано състояние на обектите и др.).

Например ако имаме метод, който прочита съдържанието на даден файл, той трябва да се казва `ReadFileContents()` и трябва да връща `byte[]` или `string` (в зависимост дали говорим за двоичен или текстов файл). Ако файлът не съществува или не може да бъде отворен по някаква причина, методът трябва да хвърли изключение, а не да върне празен низ или `null`. Връщането на неутрална стойност (например `null`) вместо съобщение за грешка не е препоръчителна практика, защото извикващият метод няма възможност да обработи грешката и изгубва носещото богата информация изключение.

⚠	**Един публичен метод или трябва да върши коректно точно това, което предполага името му, или трябва да съобщава за грешка. Всякакво друго поведение е некоректно.**

Описаното правило има някои изключения. Обикновено то се прилага най-вече за публичните методи в класа. Те или трябва да работят коректно, или трябва да съобщят за грешка. При скритите (`private`) методи може да се направи компромис - да не се проверява за некоректни параметри, тъй като тези методи може да ги извика само авторът на класа, а той би трябвало добре знае какво подава като параметри и не винаги трябва да обработва изключителните ситуации, защото може да ги предвиди. Но не забравяйте – това е компромис.

Ето два примера за качествени методи:

```
long Sum(int[] elements)
{
   long sum = 0;
   foreach (int element in elements)
   {
      sum = sum + element;
   }
   return sum;
}
```

```
double CalcTriangleArea(double a, double b, double c)
{
   if (a <= 0 || b <= 0 || c <= 0)
   {
      throw new ArgumentException("Sides should be positive.");
   }
   double s = (a + b + c) / 2;
   double area = Math.Sqrt(s * (s - a) * (s - b) * (s - c));
   return area;
}
```

Strong Cohesion и Loose Coupling

Правилата за логическа свързаност на отговорностите (strong cohesion) и за функционална независимост и минимална обвързаност с останалите методи и класове (loose coupling) важат с пълна сила за методите.

Вече обяснихме, че един метод трябва да решава един проблем, не няколко. Един метод не трябва да има странични ефекти или да решава няколко несвързани задачи, защото няма да можем да му дадем подходящо име, което пълно и точно го описва. Това означава, че всички методи, които пишем, трябва да имат strong cohesion, т.е. да са насочени към решаването на една единствена задача.

Методите трябва минимално да зависят от останалите методи и от класа, в който се намират и от останалите класове. Това свойство се нарича loose coupling.

В идеалния случай даден метод трябва да зависи единствено от параметрите си и да не използва никакви други данни като вход или като изход. Такива методи лесно могат да се извадят и да се преизползват в друг проект, защото са независими от средата, в която се изпълняват.

Понякога методите зависят от **`private`** променливи в класа, в който са дефинирани или променят състоянието на обекта, към който принадлежат. Това не е грешно и е нормално. В такъв случай говорим за обвързване (coupling) между метода и класа. Такова обвързване не е проблемно, защото целият клас може да се извади и премести в друг проект и ще започне да работи без проблем. Повечето класове от Common Type System дефинират методи, които зависят единствено от данните в класа, който ги дефинира и от подадените им параметри. В стандартните библиотеки зависимостите на методите от външни класове са минимални и затова тези библиотеки са лесни за използване.

Ако даден метод чете или променя глобални данни или зависи от още 10 обекта, които трябва да се инициализирани в инстанцията на неговия клас, той е силно обвързан с всички тези обекти. Това означава, че функционира сложно и се влияе от прекалено много външни условия и следователно възможността за грешки е голяма. Методи, които разчитат на прекалено много външни зависимости, са трудни за четене, за разбиране и за поддръжка. Силното функционално обвързване е лошо и трябва да се избягва, доколкото е възможно, защото води до код като спагети.

Сега погледнете същите два метода. Намирате ли грешки?

```
long Sum(int[] elements)
{
   long sum = 0;
   for (int i = 0; i < elements.Length; i++)
   {
      sum = sum + elements[i];
      elements[i] = 0; // Hidden side effect
   }
   return sum;
}
```

```
double CalcTriangleArea(double a, double b, double c)
{
   if (a <= 0 || b <= 0 || c <= 0)
   {
      return 0; // Incorrect result
   }
   double s = (a + b + c) / 2;
   double area = Math.Sqrt(s * (s - a) * (s - b) * (s - c));
   return area;
}
```

Колко дълъг трябва да е един метод?

През годините са правени различни изследвания за оптималната дължина на методите, но в крайна сметка универсална формула за дължина на даден метод не съществува.

Практиката показва, че като цяло трябва да предпочитаме по-кратки методи (не повече от един екран). Те са по-лесни за четене и разбиране, а вероятността да допуснем грешка при тях е значително по-малка.

Колкото по-голям е един метод, толкова по-сложен става той. Последващи модификации са значително по-трудни, отколкото при кратките методи и изискват много повече време. Тези фактори са предпоставка за допускане на грешки и по-трудна поддръжка.

Препоръчителната дължина на един метод е не-повече от един екран, но тази препоръка е само ориентировъчна. Ако методът се събира на екрана, той е по-лесен за четене, защото няма да се налага скролиране. Ако методът е по-дълъг от един екран, това трябва да ни накара да се замислим дали не можем да го разделим логически на няколко по-прости метода. Това не винаги е възможно да се направи по смислен начин, така че препоръката за дължината на методите е ориентировъчна.

Макар дългите методи да не са за предпочитане, това не трябва да е безусловна причина да разделяме на части даден метод само защото е дълъг. Методите трябва да са толкова дълги, колкото е необходимо.

> ⚠ **Силната логическа свързаност на отговорностите при методите е много по-важна от дължината им.**

Ако реализираме сложен алгоритъм и в последствие се получи дълъг метод, който все пак прави едно нещо и го прави добре, то в този случай дължината не е проблем.

Във всеки случай, винаги, когато даден метод стане прекалено дълъг, трябва да се замисляме, дали не е по-подходящо да изнесем част от кода в отделни методи, изпълняващи определени подзадачи.

Параметрите на методите

Едно от основните правила за подредба на параметрите на методите е основният или основните параметри да са първи. Пример:

```
public void Archive(PersonData person, bool persistent) {...}
```

Обратното би било доста по-объркващо:

```
public void Archive(bool persistent, PersonData person) {...}
```

Друго основно правило е имената на параметрите да са смислени. Честа грешка, е имената на параметрите да бъдат свързани с имената на типовете им. Пример:

```
public void Archive(PersonData personData) {...}
```

Вместо нищо незначещото име `personData` (което носи информация единствено за типа), можем да използваме по-добро име (така е доста по-ясно кой точно обект архивираме):

```
public void Archive(PersonData loggedUser) {...}
```

Ако има методи с подобни параметри, тяхната подредба трябва да е консистентна. Това би направило кода много по-лесен за четене:

```
public void Archive(PersonData person, bool persistent) {...}

public void Retrieve(PersonData person, bool persistent) {...}
```

Важно е да няма параметри, които не се използват. Те само могат да подведат ползвателя на този код.

Параметрите не трябва да се използват и като работни променливи – не трябва да се модифицират. Ако модифицирате параметрите на методите, кодът става по-труден за четене и логиката му – по-трудна за проследяване. Винаги можете да дефинирате нова променлива вместо да променяте параметър. Пестенето на памет не е оправдание в този сценарий.

Неочевидните допускания трябва да се документират. Например мерната единица при подаване на числа. Ако имаме метод, който изчислява косинус от даден ъгъл, трябва да документираме дали ъгълът е в градуси или в радиани, ако това не е очевидно.

Броят на параметрите не трябва да надвишава 7. Това е специално, магическо число. Доказано е, че човешкото съзнание не може да следи повече от около 7 неща едновременно. Разбира се, тази препоръка е само за ориентир. Понякога се налага да предавате и много повече параметри. В такъв случай се замислете дали не е по-добре да ги предавате като някакъв клас с много полета. Например ако имате метода **`AddStudent(…)`** с 15 параметъра (име, адрес, контакти и още много други), можете да намалите параметрите му като подавате групи логически свързани параметри като клас, примерно така: **`AddStudent(personalData, contacts, universityDetails)`**. Всеки от новите 3 параметъра ще съдържа по няколко полета и пак ще се прехвърля същата информация, но в по-лесен за възприемане вид.

Понякога е логически по-издържано вместо един обект на метода да се подадат само едно или няколко негови полета. Това ще зависи най-вече от това дали методът трябва да знае за съществуването на този обект или не. Например имаме метод, който изчислява средния успех на даден студент – **`CalcAverageResults(Student s)`**. Понеже успехът се изчислява от оценките на студента и останалите му данни нямат значение, е по-добре вместо **`Student`** да се предава като параметър списък от оценки. Така методът придобива вида **`CalcAverageResults(IList<Mark>)`**.

Правилно използване на променливите

В настоящия параграф ще разгледаме няколко добри практики при локалната работа с променливи.

Връщане на резултат

Когато връщаме резултат от метод, той трябва да се запази в променлива преди да се върне. Следният пример не казва какво се връща като резултат:

```
return days * hoursPerDay * ratePerHour;
```

По-добре би било така:

```
int salary = days * hoursPerDay * ratePerHour;
return salary;
```

Има няколко причини да запазваме резултата преди да го видим. Едната е, че така документираме кода – по името на допълнителната променлива става ясно какво точно връщаме. Другата причина е, че когато дебъгваме програмата, ще можем да я спрем в момента, в който е изчислена връщаната стойност и ще можем да проверим дали е коректна. Третата причина е, че избягваме сложните изрази, които понякога може да са няколко реда дълги и заплетени.

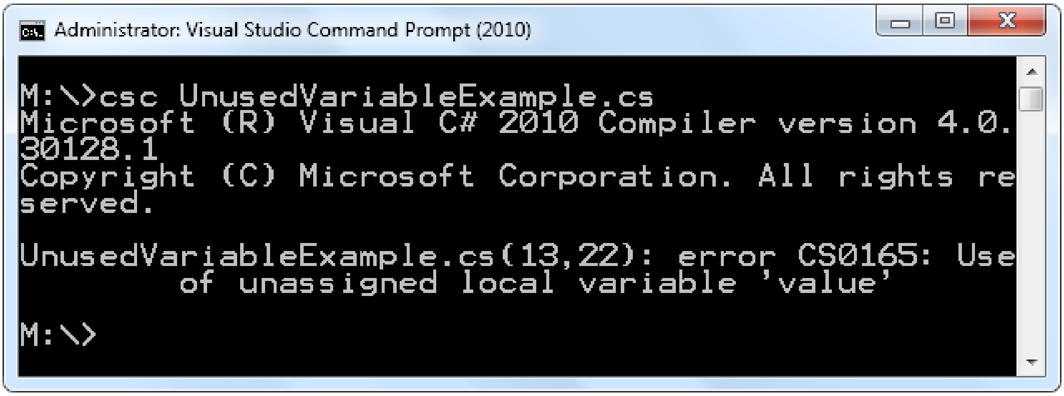
Принципи при инициализиране

В .NET всички член-променливи в класовете се инициализират автоматично още при деклариране (за разлика от C/C++). Това се извършва от средата за изпълнение. Така се избягват грешки с неправилно инициализирана памет. Всички променливи, сочещи обекти (reference type variable) се инициализират с **`null`**, а всички примитивни типове – с 0 (**`false`** за **`bool`**).

Компилаторът задължава всички локални променливи в кода на една програма да бъдат инициализирани изрично преди употреба, иначе връща грешка при компилация. Ето един пример, който ще предизвика грешка при компилация, защото се прави опит за използване на неинициализирана променлива:

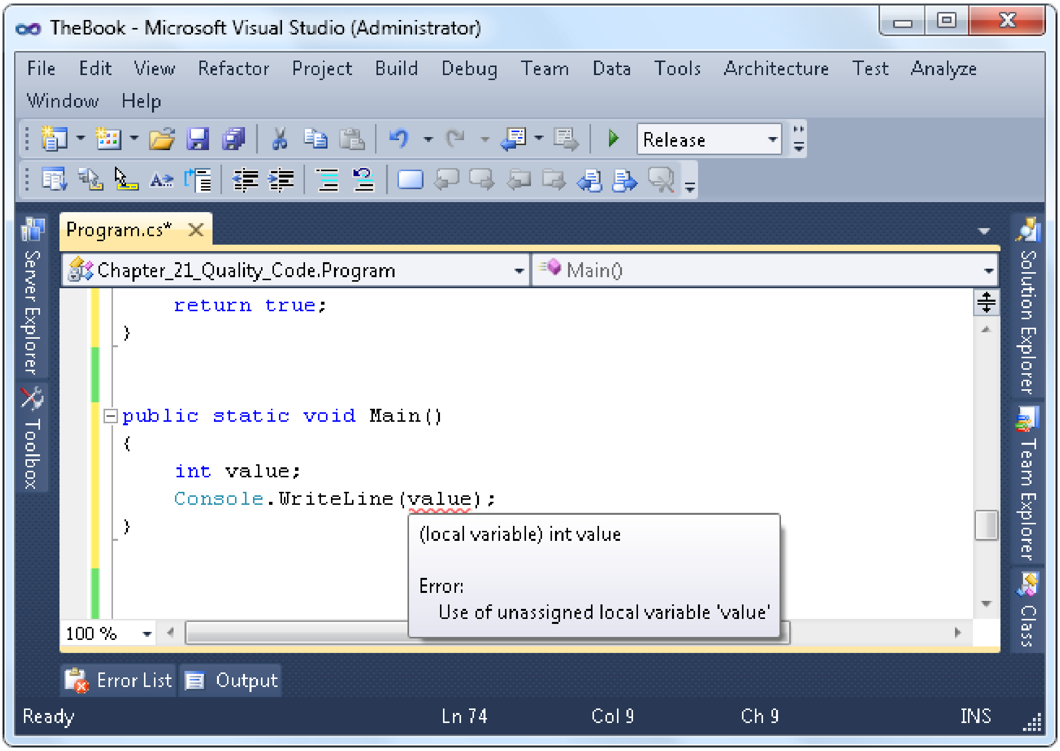
```
public static void Main()
{
   int value;
   Console.WriteLine(value);
}
```

При опит за компилация се връща грешка на втория ред:


Administrator: Visual Studio Command Prompt (2010)

```
M:\>csc UnusedVariableExample.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.
30128.1
Copyright (C) Microsoft Corporation. All rights re
served.

UnusedVariableExample.cs(13,22): error CS0165: Use
        of unassigned local variable 'value'

M:\>
```

Ето как изглеждат нещата в средата за разработка:


TheBook - Microsoft Visual Studio (Administrator)
File
Edit
View
Refactor
Project
Build
Debug
Team
Data
Tools
Architecture
Test
Analyze
Window
Help
Release
Server Explorer
Toolbox
Program.cs*
Chapter_21_Quality_Code.Program
Main()
Solution Explorer
Team Explorer
Class
100 %
Error List
Output
Ready
Ln 74
Col 9
Ch 9
INS

```
        return true;
    }


    public static void Main()
    {
        int value;
        Console.WriteLine(value);
    }
```

(local variable) int value

Error:
Use of unassigned local variable 'value'

Ето още един малко по-сложен пример:

```
int value;
if (condition1)
{
   if (condition2)
   {
      value = 1;
   }
}
else
{
   value = 2;
}
Console.WriteLine(value);
```

За щастие компилаторът е достатъчно интелигентен и хваща подобни "недоразумения" – отново същата грешка.

Забележете следната особеност: ако сложим **`else`** на вложения **`if`** в горния код, всичко ще се компилира. Компилаторът проверява всички възможни пътища, по които може да мине изпълнението и ако при всеки един от тях има инициализация на променливата, той не връща грешка и променливата се инициализира правилно.

Добрата практика е всички променливи да се инициализират изрично още при деклариране:

```
int value = 0;
Student intern = null;
```

Частично-инициализирани обекти

Някои обекти, за да бъдат правилно инициализирани, трябва да имат стойности на поне няколко техни полета. Например обект от тип **`Човек`**, трябва да има стойност на полетата "име" и "фамилия". Това е проблем, от който компилаторът не може да ни опази.

Единият начин да бъде решен този проблем е да се премахне конструкторът по подразбиране (конструкторът без параметри) и на негово място да се сложат един или няколко конструктора, които получават достатъчно данни (във формата на параметри) за правилното инициализиране на съответния обект. Точно това е идеята на такива конструктори.

Деклариране на променлива в блок/метод

Съгласно конвенцията за писане на код на .NET, една променлива трябва да се декларира в началото на блока или тялото на метода, в който се намира:

```
static int Archive()
{
   int result = 0;        // beginning of method body
   // .. Code ...
}
```

```
if (condition)
{
   int result = 0;      // beginning of an "if" block
   // .. Code ...
}
```

Изключение правят променливите, които се декларират в инициализиращата част на **`for`** цикъла:

```
for (int i = 0; i < data.Length; i++) {...}
```

Повечето добри програмисти предпочитат да декларират една променлива максимално близо до мястото, на което тя ще бъде използвана и по този начин да намалят нейния живот (погледнете следващия параграф) и същевременно възможността за грешка.

Обхват, живот, активност

Понятието **обхват на променлива** (**variable scope**) всъщност описва колко "известна" е една променлива. В .NET тя може да бъде (подредени в низходящ ред) статична променлива, член-променлива (на клас) и локална променлива (в метод).

Колкото по-голям е обхватът на дадена променлива, толкова по-голяма е възможността някой да се обвърже с нея и така да увеличи своя coupling, което не е хубаво. Следователно обхватът на променливите трябва да е възможно най-малък.

Добър подход при работата с променливи е първоначално те да са с минимален обхват. При необходимост той да се разширява. Така по естествен начин всяка променлива получава необходимия за работата й обхват. Ако не знаете какъв обхват да ползвате, започвайте от **`private`** и при нужда преминавайте към **`protected`** или **`public`**.

Статичните променливи е най-добре да са винаги **`private`** и достъпът до тях да става контролирано, чрез извикване на подходящи методи.

Ето един пример за лошо семантично обвързване със статична променлива – ужасно лоша практика:

```
public class Globals
{
   public static int state = 0;
}

public class Genious
{
   public static void PrintSomething()
   {
    if (Globals.state == 0)
      Console.WriteLine("Hello.");
    else
      Console.WriteLine("Good bye.");
  }
}
```

Ако променливата **`state`** беше дефинирана като **`private`**, такова обвързване нямаше да може да се направи, поне не директно.

Диапазон на активност (**span**) е средният брой линии между обръщенията към дадена променлива. Той зависи от гъстотата на редовете код, в които тази променлива се използва. Диапазонът на променливите трябва да е минимален. По тази причина променливите трябва да се декларират и инициализират възможно най-близко до мястото на първата им употреба, а не в началото на даден метод или блок.

Живот (**lifetime**) на една променлива е обемът на кода от първото до последното ѝ рефериране в даден метод. В тази дефиниция имаме предвид само локални променливи, понеже член-променливите живеят докато съществува класът, в който са дефинирани, а статичните променливи – докато съществува виртуалната машина.

Ето един пример за неправилно използване на променливи (излишно голям диапазон на активност):

```
int count;
int[] numbers = new int[100];

for (int i = 0; i < numbers.Length; i++)
{
  numbers[i] = i;
}
count = 0;

for (int i = 0; i < numbers.Length / 2; i++)
{
  numbers[i] = numbers[i] * numbers[i];
}

for (int i = 0; i < numbers.Length; i++)
{
  if (numbers[i] % 3 == 0)
  {
    count++;
  }
}

Console.WriteLine(count);
```

lifetime = 23 lines

span = 23 / 4 = 5.75

В този пример променливата `count` служи за преброяване на числата, които се делят без остатък на 3 и се използва само в последния `for` цикъл. Тя е дефинирана излишно рано и се инициализира много преди да има нужда от инициализацията. Ако трябва да се преработи този код, за да се намали диапазонът на активност на променливата `count`, той ще добие следния вид:

```
int[] numbers = new int[100];
for (int i = 0; i < numbers.Length; i++)
{
  numbers[i] = i;
}

for (int i = 0; i < numbers.Length / 2;
i++)
{
  numbers[i] = numbers[i] * numbers[i];
}

int count = 0;
for (int i = 0; i < numbers.Length; i++)
{
  if (numbers[i] % 3 == 0)
  {
    count++;
  }
}

Console.WriteLine(count);
```

lifetime = 10 lines
span = 10 / 3 = 3.33

Важно е програмистът да следи къде се използва дадена променлива, нейният диапазон на активност и период на живот. Основното правило е да се направят обхватът, животът и активността на променливите колкото се може по-малки. От това следва едно важно правило:

Декларирайте локалните променливи възможно най-късно, непосредствено преди да ги използвате за първи път, и ги инициализирайте заедно с декларацията им.

Променливите с по-голям обхват и по-дълъг живот, трябва да имат по-описателни имена, примерно `totalStudentsCount`. Причината е, че те ще бъдат използвани на повече места и за по-дълго време и за какво служат няма да бъде ясно от контекста. Променливите с живот няколко реда могат да бъдат с кратко и просто име, примерно `count`. Те нямат нужда от дълги и описателни имена, защото техният смисъл е ясен от контекста, в който се използват, а този контекст е твърде малък (няколко реда), за да има двусмислия.

Работа с променливи – още правила

Една променлива трябва да се използва само за една цел. Това е много важно правило. Извиненията, че ако се преизползва едно променлива за няколко цели се пести на памет, в общия случай не са добро оправдание. Ако една променлива се ползва за няколко съвсем различни цели, какво име ще й дадем? Например, ако една променлива се използва да брои студенти и в някои случаи техните оценки, то как ще я кръстим: **`count`**, **`studentsCount`**, **`marksCount`** или **`StudentsOrMarksCount`**?

Ползвайте една променлива само за една единствена цел. Иначе няма да можете да й дадете подходящо име.

Никога не трябва да има променливи, които не се използват. В такъв случай тяхното дефиниране е било безсмислено. За щастие сериозните среди за разработка издават предупреждение за подобни "нередности".

Трябва да се избягват и променливи със скрито значение. Например Пешо е оставил променливата **`x`**, за да бъде видяна от Митко, който трябва да се сети да имплементира още един метод, в който ще я ползва.

Правилно използване на изрази

При работата с изрази има едно много просто правило: не ползвайте сложни изрази! Сложен израз наричаме всеки израз, който извършва повече от едно действие. Ето пример за сложен израз:

```
for (int i = 0; i < xCoord.Length; i++)
{
  for (int j = 0; j < yCoord.Length; j++)
  {
    matrix[i][j] =
      matrix[xCoord[FindMax(i) + 1]][yCoord[FindMin(i) + 1]] *
      matrix[yCoord[FindMax(i) + 1]][xCoord[FindMin(i) + 1]];
  }
}
```

В примерния код имаме сложно изчисление, което запълва дадена матрица спрямо някакви изчисления върху някакви координати. Всъщност е много трудно да се каже какво точно се случва, защото е използван сложен израз.

Има много причини, заради които трябва да избягваме използването на сложни изрази като в примера по-горе. Ще изброим някои от тях:

- Кодът трудно се чете. В нашия пример няма да ни е лесно да разберем какво прави този код и дали е коректен.

- Кодът трудно се поддържа. Помислете, какво ще ни струва да поправим грешка в този код, ако не работи коректно.
- Кодът трудно се поправя, ако има дефекти. Ако примерният код по-горе даде **`IndexOutOfRangeException`**, как ще разберем извън границите на кой точно масив сме излезли? Това може да е масивът **`xCoord`** или **`yCoord`** или **`matrix`**, а излизането извън тези масиви може да е на няколко места.
- Кодът трудно се дебъгва. Ако намерим грешка, как ще дебъгнем изпълнението на този израз, за да намерим грешката?

Всички тези причини ни подсказват, че писането на сложни изрази е вредно и трябва да се избягва. Вместо един сложен израз можем да напишем няколко по-прости изрази и да ги запишем в променливи с разумни имена. По този начин кодът става по-прост, по-ясен, по-лесен за четене и разбиране, по-лесен за промяна, по-лесен за дебъгване и по-лесен за поправяне. Нека сега пренапишем горния код, без да използваме сложни изрази:

```
for (int i = 0; i < xCoord.Length; i++)
{
   for (int j = 0; j < yCoord.Length; j++)
   {
      int maxStartIndex = FindMax(i) + 1;
      int minStartIndex = FindMax(i) - 1;
      int minXcoord = xCoord[minStartIndex];
      int maxXcoord = xCoord[maxStartIndex];
      int minYcoord = yCoord[minStartIndex];
      int maxYcoord = yCoord[maxStartIndex];
      matrix[i][j] =
         matrix[maxXcoord][minYcoord] *
         matrix[maxYcoord][minXcoord];
   }
}
```

Забележете колко по-прост и ясен стана кода. Наистина, без да знаем какво точно изчисление извършва този код, ще ни е трудно да го разберем, но ако настъпи изключение, лесно ще намерим на кой ред възниква и чрез дебъгера можем да проследим защо се получава и евентуално да го поправим.

Не пишете сложни изрази. На един ред трябва да се извършва по една операция. Иначе кодът става труден за четене, за поддръжка, за дебъгване и за промяна.

Използване на константи

В добре написания програмен код не трябва да има "магически числа" и стрингове. Такива наричаме всички литерали в програмата, които имат стойност, различно от `0`, `1`, `-1`, `""` и **`null`** (с дребни изключения).

За да обясним по-добре концепцията за използване на именувани константи, ще дадем един пример за код, който има нужда от преработка:

```
public class GeometryUtils
{
  public static double CalcCircleArea(double radius)
  {
    double area = 3.14159206 * radius * radius;
    return area;
  }

  public static double CalcCirclePerimeter(double radius)
  {
    double perimeter = 6.28318412 * radius;
    return perimeter;
  }

  public static double CalcElipseArea(double axis1, double axis2)
  {
    double area = 3.14159206 * axis1 * axis2;
    return area;
  }
}
```

В примера използваме три пъти числото **`3.14159206`** (П), което е повторение на код. Ако решим да променим това число, като го запишем например с по-голяма точност, ще трябва да променим програмата на три места. Възниква идеята да дефинираме това число като стойност, която е глобална за програмата и не може да се променя. Именно такива стойности в .NET се декларират като именувани константи по следния начин:

```
public const double PI = 3.14159206;
```

След тази декларация константата **`PI`** е достъпна от цялата програма и може да се ползва многократно. При нужда от промяна променяме само на едно място и промените се отразяват навсякъде. Ето как изглежда нашия примерен клас **`GeometryUtils`** след изнасянето на числото **`3.14159206`** в константа:

```
public class GeometryUtils
{
  public const double PI = 3.14159206;
```

```
   public static double CalcCircleArea(double radius)
   {
      double area = PI * radius * radius;
      return area;
   }

   public static double CalcCirclePerimeter(double radius)
   {
      double perimeter = 2 * PI * radius;
      return perimeter;
   }

   public static double CalcElipseArea(
      double axis1, double axis2)
   {
      double area = PI * axis1 * axis2;
      return area;
   }
}
```

Кога да използваме константи?

Използването на константи помага да избегнем използването на "магически числа" и стрингове в нашите програми и позволява да дадем имена на числата и стринговете, които ползваме. В предходния пример не само избегнахме повторението на код, но и документирахме факта, че числото **`3.14159206`** е всъщност добре известната в математиката константа π.

Константи трябва да дефинираме винаги, когато имаме нужда да ползваме числа или символни низове, за които не е очевидно от къде идват и какъв е логическият им смисъл. Константи е нормално да дефинираме и за всяко число или символен низ, който се ползва повече от веднъж в програмата.

Ето няколко типични ситуации, в които трябва да ползвате именувани константи:

- За имена на файлове, с които програмата оперира. Те често трябва да се променят и затова е много удобно да са изнесени като константи в началото на програмата.
- За константи, участващи в математически формули и преобразувания. Доброто име на константата подобрява шансът при четене на кода да разберете смисъла на формулата.
- За размери на буфери или блокове памет. Тези размери може да се наложи да се променят и е удобно да са изнесени като константи. Освен това използването на константата **`READ_BUFFER_SIZE`** вместо някакво магическо число 8192 прави кода много по-ясен и разбираем.

Кога да не използваме константи?

Въпреки, че много книги препоръчват всички числа и символни низове, които не са `0`, `1`, `-1`, `""` и `null` да бъдат изнасяни като константи, има някои изключения, в които изнасянето на константи е вредно. Запомнете, че изнасянето на константи се прави, за да се подобри четимостта на кода и поддръжката му във времето. Ако изнасянето на дадена константа не подобрява четимостта на кода, няма нужда да го правите.

Ето някои ситуации, в които изнасянето на текст или магическо число като константа не е полезно:

- Съобщения за грешки и други съобщения към потребителя (примерно "въведете името си"): изнасянето им затруднява четенето на кода вместо да го улесни.
- SQL заявки (ако използвате бази от данни, командите за извличане на информацията от базата данни се пише на езика SQL и представлява стринг). Изнасянето на SQL заявки като константи прави четенето на кода по-трудно и не се препоръчва.
- Заглавия на бутони, диалози, менюта и други компоненти от потребителския интерфейс също не се препоръчва да се изнасят като константи, тъй като това прави кода по-труден за четене.

В .NET съществуват библиотеки, които подпомагат интернационализацията и позволяват да изнасяте съобщения за грешки, съобщения към потребителя и текстовете в потребителския интерфейс в специални ресурсни файлове, но това не са константи. Такъв подход се препоръчва, ако програмата, която пишете ще трябва да се интернационализира.

Използвайте именувани константи, за да избегнете използването и повтарянето на магически числа и стрингове в кода и най-вече, за да подобрите неговата четимост. Ако въвеждането на именувана константа затруднява четимостта на програмата, по-добре оставете твърдо зададената стойност в кода!

Правилно използване на конструкциите за управление

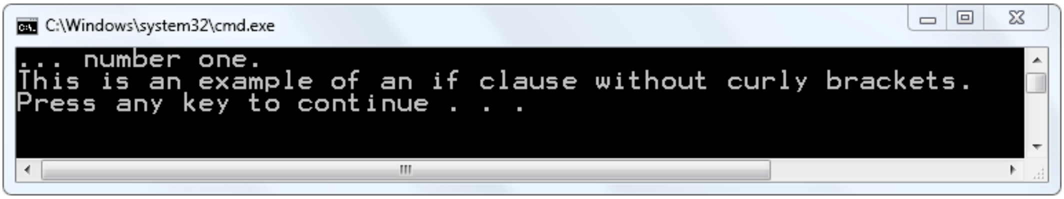
Конструкциите за управление са циклите и условните конструкции. Сега ще разгледаме добрите практики за правилното им използване.

Със или без къдрави скоби?

Циклите и условните конструкции позволяват тялото да не се огражда със скоби и да се състои от един оператор (statement). Това е опасно. Вижте следния пример:

```
static void Main()
{
  int two = 2;
  if (two == 1)
    Console.WriteLine("This is the ...");
    Console.WriteLine("... number one.");

  Console.WriteLine(
    "This is an example of an if clause without curly
brackets.");
}
```

Очакваме да се изпише само последното изречение? Резултатът е малко неочакващ:



```
... number one.
This is an example of an if clause without curly brackets.
Press any key to continue . . .
```

Появява се един допълнителен ред. Това е защото в if-клаузата влиза само първия оператор (statement) след нея. Вторият е просто неправилно подравнен и объркващ.

Винаги заграждайте тялото на циклите и условните конструкции с къдрави скоби – { и }.

Правилно използване на условни конструкции

Условни конструкции в C# са **`if-else`** операторите и **`switch-case`** операторите.

```
if (condition)
{

}
else
{

}
```

Дълбоко влагане на if-конструкции

Дълбокото влагане на if-конструкции е лоша практика, защото прави кода сложен и труден за четене. Ето един пример:

```
private int Max(int a, int b, int c, int d)
{
   if (a < b)
   {
      if (b < c)
      {
         if (c < d)
         {
            return d;
         }
         else
         {
            return c;
         }
      }
      else if (b > d)
      {
         return b;
      }
      else
      {
         return d;
      }
   }
   else if (a < c)
   {
      if (c < d)
      {
         return d;
      }
      else
      {
         return c;
      }
   }
   else if (a > d)
   {
      return a;
   }
   else
   {
      return d;
   }
}
```

Този код е напълно нечетим. Причината е, че има прекалено дълбоко влагане на **`if`** конструкциите една в друга. За да се подобри четимостта на този код, може да се въведат един или няколко метода, в които да се изнесе част от сложната логика. Ето как може да се преработи кода, за да се намали вложеността на условните конструкции и да стане по-разбираем:

```
private int Max(int a, int b)
{
   if (a < b)
   {
      return b;
   }
   else
   {
      return a;
   }
}

private int Max(int a, int b, int c)
{
   if (a < b)
   {
      return Max(b, c);
   }
   else
   {
      return Max(a, c);
   }
}

private int Max(int a, int b, int c, int d)
{
   if (a < b)
   {
      return Max(b, c, d);
   }
   else
   {
      return Max(a, c, d);
   }
}
```

Изнасянето на част от кода в отделен метод и най-лесния и ефективен начин да се намали вложеността на група условни конструкции, като се запази логическият им смисъл.

Преработеният метод е разделен на няколко по-малки. Така резултатът като цяло е с 9 реда по-малко. Всеки от новите методи е много по-прост и лесен за четене. Като страничен ефект получаваме допълнително 2 метода, които можем да използваме и за други цели.

Правилно използване на цикли

Правилното използване на различните конструкции за цикли е от значение при създаването на качествен софтуер. В следващите параграфи ще се запознаем с някои принципи, които ни помагат да определим кога и как да използваме определен вид цикъл.

Избиране на подходящ вид цикъл

Ако в дадена ситуация не можем да решим дали да използваме `for`, `while` или `do-while` цикъл, можем лесно да решим проблема, придържайки се към следващите принципи:

Ако се нуждаем от цикъл, който да се изпълни определен брой пъти, то е добре да използваме `for` цикъл. Този цикъл се използва в прости случаи, когато не се налага да прекъсваме изпълнението. При него още в началото задаваме параметрите на цикъла и в общия случай, в тялото не се грижим за контрола му. Стойността на брояча вътре в тялото на цикъла не трябва да се променя.

Ако е необходимо да следим някакви условия, при които да прекратим изпълнението на цикъла, тогава вероятно е по-добре да използваме `while` цикъл. `while` цикълът е подходящ в случаи, когато не знаем колко точно пъти трябва да се изпълни тялото цикъла. При него изпълнението продължава, докато не се достигне дадено условие за край. Ако имаме налице предпоставките за използване на `while` цикъл, но искаме да сме сигурни, че тялото ще се изпълни поне веднъж, то в такъв случай трябва да използваме `do-while` цикъл.

Не влагайте много цикли

Както и при условните конструкции, и при циклите е лоша практика да имаме дълбоко влагане. Дълбокото влагане обикновено се получава от голям брой цикли и условни конструкции, поставени една в друга. Това прави кода сложен и труден за четене и поддръжка. Такъв код лесно може да се подобри, като се отдели част от логиката в отделен метод. Съвременните среди за разработка могат да правят такава преработка на кода автоматично (ще обясним за това в секцията за преработка на кода).

Защитно програмиране

Защитно програмиране (defensive programming) е термин обозначаващ практика, която е насочена към защита на кода от некоректни данни. Защитното програмиране пази кода от грешки, които никой не очаква. То се имплементира чрез проверка на коректността на всички входни данни. Това са данните, идващи от външни източници, входните параметри на методите, конфигурационни файлове и настройки, данни въведени от потребителя, дори и данни от друг локален метод.

Защитното програмиране изисква всички данни да се проверяват, дори да идват от източник, на когото се вярва. По този начин, ако в този източник има грешка (бъг), то тя ще бъде открита по-бързо.

Защитното програмиране се имплементира чрез assertions, изключения и други средства за управление на грешки.

Assertions

Това са специални условия, които винаги трябва да са изпълнени. Неизпълнението им завършва с грешка. Ето един бърз пример:

```
void LoadTemplates(string fileName)
{
  bool templatesFileExist = File.Exists(fileName);
  Debug.Assert(templatesFileExist,
    "Can't load templates file: " + fileName);
}
```

Assertions vs. Exceptions

Изключенията са анонси за грешка или неочаквано събитие. Те информират ползвателя на кода за грешка. Изключенията могат да бъдат "хванати" и изпълнението може да продължи.

Assertions (без наложил се термин на български) са най-общо фатални грешки. Не могат да бъдат хванати или обработени. Винаги индикират бъг в кода. Приложението не може да продължи.

Assertions могат да се изключват. По замисъл те трябва да са включени само по време на разработка, докато се открият всички бъгове. Когато бъдат изключени всички проверки в тях спират да се изпълняват. Идеята на изключването е, че след края на разработката, тези проверки не са повече нужни и само забавят софтуера.

Ако дадена проверка е смислено да продължи да съществува след края на разработката (примерно проверява входни данни на метод, които идват от потребителя), то тази проверка е неправилно имплементирана с assertions и трябва да бъде имплементирана с изключения.

Assertions се използват само на места, на които трябва дадено условие да бъде изпълнено и единствената причина да не е, е да има бъг в програмата.

Защитно програмиране с изключения

Изключенията (exceptions) предоставят мощен механизъм за централизирано управление на грешки и непредвидени ситуации. В главата "Обработка на изключения" те са описани подробно.

Изключенията позволяват проблемните ситуации да се обработват на много нива. Те улесняват писането и поддръжката на надежден програмен код.

Разликата между изключенията и assertions е в това, че изключенията в защитното програмиране се използват най-вече за защитаване на публичния интерфейс на един компонент. Този механизъм се нарича **fail-safe** (в свободен превод "проваляй се грациозно" или "подготвен за грешки").

Ако методът **`archive`**, описан малко по-нагоре, беше част от публичния интерфейс на архивиращ компонент, а не вътрешен метод, то този метод би трябвало да бъде имплементиран така:

```
public int Archive(PersonData user, bool persistent)
{
   if (user == null)
      throw new StorageException("null parameter");

   // Do some processing
   int resultFromProcessing = ...

   Debug.Assert(resultFromProcessing >= 0,
      "resultFromProcessing is negative. There is a bug");

   return resultFromProcessing;
}
```

Assert остава, тъй като той е предвиден за променлива създадена вътре в метода.

Изключенията трябва да се използват, за да се уведомят другите части на кода за проблеми, които не трябва да бъдат игнорирани. Хвърлянето на изключение е оправдано само в ситуации, които наистина са изключителни и трябва да се обработят по някакъв начин. За повече информация за това кои ситуации са изключителни и кои не погледнете главата "Обработка на изключения".

Ако даден проблем може да се обработи локално, то обработката трябва да се направи в самия метод и изключение не трябва да се хвърля. Ако даден проблем не може да се обработи локално, той трябва да бъде прехвърлен към извикващия метод.

Трябва да се хвърлят изключения с подходящо ниво на абстракция. Пример: **`GetEmployeeInfo()`** може да хвърля **`EmployeeException`**, но не и **`FileNotFoundException`**. Погледнете последният пример, той хвърля **`StorageException`**, а не **`NullReferenceException`**.

Повече за добрите практики при управление на изключенията можете да прочетете от секцията "Добри практики при работа с изключения" на главата "Обработка на изключения".

Документация на кода

C# спецификацията позволява писане на коментари в кода. Вече се запознахме с основните начини на писане на коментари. В следващите няколко параграфа ще обясним как се пишат ефективни коментари.

Самодокументиращ се код

Коментарите в кода не са основният източник на документация. Запомнете това! Добрият стил на програмиране е най-добрата документация! Самодокументиращ се код е такъв, на който лесно се разбира основната му цел, без да е необходимо да има коментари.

⚠	**Най-добрата документация на кода е да пишем качествен код. Лошият код не трябва да се коментира, а трябва да се пренапише, така че сам да описва себе си. Коментарите в програмата само допълват документацията на добре написания код.**

Характеристики на самодокументиращия се код

Характеристики на самодокументиращия се код са добра структура на програмата – подравняване, организация на кода, използване на ясни и лесни за разбиране конструкции, избягване на сложни изрази. Такива са още употребата на подходящи имена на променливи, методи и класове и употребата на именувани константи, вместо "магически" константи и текстови полета. Реализацията трябва да е опростена максимално, така че всеки да я разбере.

Самодокументиращ се код – важни въпроси

Въпроси, които трябва да си зададем преди да отговорим на въпроса дали кодът е самодокументиращ се:

- Подходящо ли е името на класа и показва ли основната му цел?
- Става ли ясно от интерфейса как трябва да се използва класа?
- Показва ли името на метода основната му цел?
- Всеки метод реализира ли една добре определена задача?
- Имената на променливите съответстват ли на тяхната употреба?
- Групирани ли са свързаните един с друг оператори?
- Само една задача ли изпълняват конструкциите за итерация (циклите)?
- Има ли дълбоко влагане на условни конструкции?
- Показва ли организацията на кода неговата логическата структура?

- Дизайнът недвусмислен и ясен ли е?
- Скрити ли са детайлите на имплементацията възможно най-много?

"Ефективни" коментари

Коментарите понякога могат да навредят повече, отколкото да помогнат. Добрите коментари не повтарят кода и не го обясняват – те изясняват неговата идея. Коментарите трябва да обясняват на по-високо ниво какво се опитваме да постигнем. Писането на коментари помага да осмислим по-добре това, което искаме да реализираме.

Ето един пример за лоши коментари, които повтарят кода и вместо да го направят по-лесно четим, го правят по-тежък за възприемане:

```
public List<int> FindPrimes(int start, int end)
{
   // Create new list of integers
   List<int> primesList = new List<int>();
   // Perform a loop from start to end
   for (int num = start; num <= end; num++)
   {
      // Declare boolean variable, initially true
      bool prime = true;
      // Perform loop from 2 to sqrt(num)
      for (int div = 2; div <= Math.Sqrt(num); div++)
      {
         // Check if div divides num with no remainder
         if (num % div == 0)
         {
            // We found a divider -> the number is not prime
            prime = false;
            // Exit from the loop
            break;
         }
         // Continue with the next loop value
      }

      // Check if the number is prime
      if (prime)
      {
         // Add the number to the list of primes
         primesList.Add(num);
      }
   }

   // Return the list of primes
   return primesList;
}
```

Ако вместо да слагаме наивни коментари, ги ползваме, за да изясним неочевидните неща в кода, те могат да са много полезни. Вижте как бихме могли да коментираме същия код, така че да му подобрим четимостта:

```
/// <summary>
/// Finds primes from a range [a,b] and returns them in a list.
/// </summary>
/// <param name="start">Top of range</param>
/// <param name="end">End of range</param>
/// <returns>
/// a list of all the found primes
/// </returns>
public List<int> FindPrimes(int start, int end)
{
   List<int> primesList = new List<int>();
   for (int num = start; num <= end; num++)
   {
      bool isPrime = IsPrime(num);
      if (isPrime)
      {
         primesList.Add(num);
      }
   }
   return primesList;
}

/// <summary>
/// Checks if a number is prime by checking for any
/// dividers in the range [2, sqrt(number)].
/// </summary>
/// <param name="number">The number to be checked</param>
/// <returns>True if prime</returns>
public bool IsPrime(int number)
{
   for (int div = 2; div <= Math.Sqrt(number); div++)
   {
      if (number % div == 0)
      {
         return false;
      }
   }

   return true;
}
```

Логиката на кода е очевидна и няма нужда от коментари. Достатъчно е да се опише за какво служи даденият метод и основната му идея (как работи) в едно изречение.

При писането на "ефективни" коментари е добра практика да се използва псевдокод, когато е възможно. Коментарите трябва да се пишат, когато се създава самия код, а не след това.

Продуктивността никога не е добра причина, за да не се пишат коментари. Трябва да се документира всичко, което не става ясно от кода. Поставянето на излишно много коментари е толкова вредно колкото и липсата на такива.

Лошият код не става по-добър с повече коментари. За да стане добър код, просто трябва да се преработи.

Преработка на кода (Refactoring)

Терминът Refactoring се появява през 1993 и е популяризиран от Мартин Фаулър в едноименната му книга по темата. В тази книга се разглеждат много техники за преработка на код. Нека и ние разгледаме няколко.

Дадена програма се нуждае от преработка, при повторение на код. Повторението на код е опасно, защото когато трябва да се променя, трябва да се променя на няколко места и естествено някое от тях може да бъде пропуснато и така да се получи несъответствие. Избягването на повтарящ се код може да стане чрез изваждане на метод или преместване на код от клас-наследник в базов клас.

Преработка се налага и при методи, които са нараснали с времето. Прекалената дължината на метод е добра причина да се замислим дали методът не може да се раздели логически на няколко по-малки и по-прости метода.

При цикъл с прекалено дълбоко ниво на влагане трябва да се замислим дали не можем да извадим в отделен метод част от кода му. Обикновено това подобрява четимостта на кода и го прави по-лесен за разбиране.

Преработката е наложителна при клас, който изпълнява несвързани отговорности (weak cohesion). Клас, който не предоставя достатъчно добро ниво на абстракция също трябва да се преработи.

Дългият списък с параметри и публичните полета също трябва да са в графата "да се поправи". Тази графа трябва да допълни и когато една промяна налага да се променят паралелно още няколко класа. Прекалено свързани класове или недостатъчно свързани класове също трябва да се преработят.

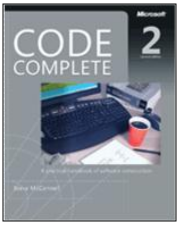
Преработка на код на ниво данни

Добра практика е в кода да няма "магически" числа. Те трябва да бъдат заменени с константи. Променливите с неясни имена трябва да се преименуват. Дългите условни изрази могат да бъдат преработени в отделни методи. За резултата от сложни изрази могат да се използват междинни променливи. Група данни, които се появяват заедно могат да се преботят в отделен клас. Свързаните константи е добре да се преместят в изброими типове (enumerations).

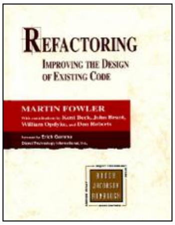
Добра практика е всички задачи от един по-голям метод, които не са свързани с основната му цел, да се "преместят" в отделни методи (extract method). Сходни задачи трябва да се групират в общи класове, сходните класове – в общ пакет. Ако група класове имат обща функционалност, то тя може да се изнесе в базов клас.

Не трябва да има циклични зависимости между класовете – те трябва да се премахват. Най-често по-общият клас има референция към по-специализирания (връзка родител-деца).

Ресурси



Библията за качествен програмен код се казва "Code Complete" и през 2004 година излезе във второ издание. Авторът й Стийв Макконъл е световноизвестен експерт по писане на качествен софтуер. В книгата можете да откриете много повече примери и детайлни описания на различни проблеми, които не успяхме да разгледаме.

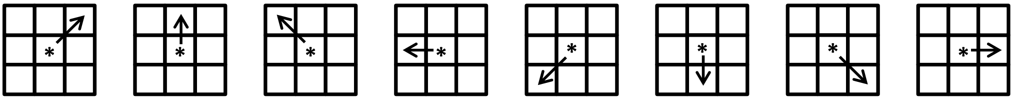


Друга добра книга е "Refactoring" на Мартин Фаулър. Тази книга се смята за библията в преработката на код. В нея за първи път са описани понятията "extract method" и други, стоящи в основата на съвременните шаблони за преработка на съществуващ код.

Упражнения

1. Вземете кода от първия пример в тази глава и го направете качествен.
2. Прегледайте собствения си код досега и вижте какви грешки допускате. Обърнете особено внимание на тях и помислете защо ги допускате. Постарайте се в бъдеще да не правите същите грешки.
3. Отворете чужд код и се опитайте само на базата на кода и документацията да разберете какво прави той. Има ли неща, които не ви стават ясни от първия път? А от втория? Какво бихте променили в този код? Как бихте го написали вие?
4. Разгледайте класове от CTS. Намирате ли примери за некачествен код?
5. Използвали ли сте (виждали ли сте) някакви код конвенции. През призмата на тази глава смятате ли, че са добри или лоши?

6. Дадена е квадратна матрица с големина **n** x **n** клетки. Въртящо обхождане на матрица наричаме такова обхождане, което започва от най-горната най-лява клетка на матрицата и тръгва към най-долната дясна. Когато обхождането не може да продължи в текущата посока (това може да се случи, ако е стигнат краят на матрицата или е достигната вече обходена клетка) посоката се сменя на следващата възможна по часовниковата стрелка. Осемте възможни посоки са:



Когато няма свободна празна клетка във всички възможни посоки, обхождането продължава от първата свободна клетка с възможно най-малък ред и възможно най-близко до началото на този ред. Обхождането приключва, когато няма свободна празна клетка в цялата матрица. Задачата е да се напише програма, която чете от конзолата цяло число **n** (1 ≤ **n** ≤ 100) и изписва запълнената матрица също на конзолата.

Примерен вход:

```
n = 6
```

Примерен изход:

```
 1  16  17  18  19  20
15   2  27  28  29  21
14  31   3  26  30  22
13  36  32   4  25  23
12  35  34  33   5  24
11  10   9   8   7   6
```

Вашата задача е да свалите от този адрес решение на горната задача: http://introcsharpbook.googlecode.com/svn/trunk/book/resources/High-Quality-Code.rar и да го преработите според концепциите за качествен код. Може да ви се наложи да оправяте и бъгове в решението.

Решения и упътвания

1. Използвайте [Ctrl+K, Ctrl+F] във Visual Studio или C# Developer и вижте разликите. След това отново с помощта на средата преименувайте променливите, премахнете излишните оператори и променливи и направете текста, който се отпечатва на екрана по-смислен.

2. Внимателно следвайте препоръките за конструиране на качествен програмен код от настоящата тема. Записвайте грешките, които правите най-често и се постарайте да ги избягвате.

3. Вземете като пример някой качествено написан софтуер. Вероятно ще откриете неща, които бихте написали по друг начин или тази глава съветва да се напишат по друг начин. Отклоненията са възможни и са

съвсем нормални. Разликата между качествения и некачествения софтуер е в последователността на спазване на правилата.

4. Кодът от CTS е писан от инженери с дългогодишен опит и в него рядко ще срещнете некачествен код. Въпреки всичко се срещат недоразумения като използване на сложни изрази, неправилно именувани променливи и други.
5. Разгледайте код, кой вие или ваши колеги са писали.
6. Прегледайте всички изучени концепции и ги приложете върху дадения код. Първо оправете кода, осмислете как работи и чак тогава оправете бъговете, които откриете при неговата работа.

Telerik Academy

Присъедини се към Академията на Телерик!

АКАДЕМИЯТА НА ТЕЛЕРИК предоставя безплатно практическо обучение, насочено към всички млади хора, желаещи да станат умели .NET софтуерни инженери и да се присъединят към екипа на Телерик.

Академията подготвя бъдещите софтуерни специалисти за сложността, разнообразието и динамиката на днешната ИТ действителност, като за целта всички курсисти се обучават да работят с най-новите технологии на Майкрософт и черпят знания и опит от доказали се в областта на програмирането лектори и разработчици на софтуер.

В академията ще получите задълбочени знания и опит, изучавайки:

C#, .NET технологии, бази данни, SQL, Silverlight, WPF, ASP.NET, HTML5, разработка на мобилни приложения за iOS, Android и Windows Phone, основите на софтуерното инженерство

Академията на Телерик ви дава възможност да:

- **Учите напълно БЕЗПЛАТНО**
- **Изберете сред редица РАЗЛИЧНИ КУРСОВЕ**
- **Овладеете ОСНОВИТЕ на софтуерното инженерство**
- **Усвоите ПРОЦЕСА за разработка на софтуер**
- **Получите задълбочени теоретични и практически ИТ ПОЗНАНИЯ**
- **Станете умел .NET СОФТУЕРЕН ИНЖЕНЕР**
- **Започнете своята ИТ кариера в ТЕЛЕРИК – РАБОТОДАТЕЛ #1 в България за 2010 г.**

Само в рамките на две години **АКАДЕМИЯТА НА ТЕЛЕРИК** за софтуерни инженери успя да се наложи като безспорен лидер у нас в предлагането на допълнително обучение за софтуерни специалисти, спомагайки за успешния старт в кариерното развитие на стотици ентусиазирани младежи.

academy.telerik.com

academy@telerik.com *facebook.com/TelerikAcademy*

telerik

deliver more than expected

Глава 22. Ламбда изрази и LINQ заявки

В тази тема...

В настоящата тема ще се запознаем с част от по-сложните възможности на езика C# и по-специално ще разгледаме как се правят заявки към колекции чрез ламбда изрази и LINQ заявки. Ще обясним как да добавяме функционалност към съществуващи вече класове, използвайки разширяващи методи (extension methods). Ще се запознаем с анонимните типове (anonymous types), ще опишем накратко какво представляват и как се използват. Ще разгледаме ламбда изразите (lambda expressions), ще покажем с примери как работят повечето вградени ламбда функции. След това ще обърнем по-голямо внимание на синтаксиса на LINQ. Ще научим какво представлява, как работи и какви заявки можем да конструираме с него. Накрая ще се запознаем с ключовите думи за LINQ, тяхното значение и ще ги демонстрираме, чрез голям брой примери.

Разширяващи методи (extension methods)

Често пъти в практиката на програмистите им се налага да добавят функционалност към вече съществуващ код. Ако кодът е наш, можем просто да добавим нужната функционалност и да прекомпилираме. Когато дадено асембли (`.exe` или `.dll` файл) е вече компилирано, и кодът не е наш, класическият вариант за разширяване на функционалността на типовете е чрез наследяване. Този подход може да стане доста сложен за осъществяване, тъй като навсякъде където се използват променливи от базовия тип, ще трябва да използваме променливи от наследяващия за да можем да достъпим нашата нова функционалност. За съжаление съществува и по-сериозен проблем. Ако типът, който искаме да наследим е маркиран с ключовата дума `sealed`, то опция за наследяване няма. **Разширяващите методи (extension methods)** решават точно този проблем – дават ни възможност да добавяме функционалност към съществуващ тип (клас или интерфейс), без да променяме оригиналния му код и дори без наследяване, т.е. работи също и с типове, които не подлежат на наследяване. Забележете, че чрез extension methods, можем да добавяме "имплементирани методи" дори към интерфейси.

Разширяващите методи се дефинират като статични методи в обикновени статични класове. Типа на първият им аргумент представлява класа (или интерфейса) към който се закачат. Преди него се слага ключовата дума `this`. Това ги отличава от другите статични методи и показва на компилатора, че това е разширяващ метод. Параметъра, пред който стои ключовата дума `this`, може да бъде използван в тялото на метода, за да се създаде функционалността на метода. Той реално представлява обекта, с който разширяващия метод работи.

Разширяващите методи могат да бъдат използвани директно върху обекти от класа/интерфейса, който разширяват. Могат да бъдат извиквани и статично чрез статичния клас в който са дефинирани, но това не е препоръчителна практика.

⚠	**За да могат да бъдат достъпени дадени разширяващи методи, трябва да бъде добавен с using съответния namespace, в който е дефиниран статичния клас описващ тези методи. В противен случай, компилаторът няма как да разбере за тяхното съществуване.**

Разширяващи методи – примери

Нека вземем за пример разширяващ метод, който брои колко думи има в даден текст (`string`). Забележете, че типа `string` е `sealed` и не може да се наследява.

```
public static class StringExtensions
{
   public static int WordCount(this string str)
   {
      return str.Split(new char[] { ' ', '.', '?', '!'  },
        StringSplitOptions.RemoveEmptyEntries).Length;
   }
}
```

Методът **WordCount()** се закача за класа **string**. Това е оказано с ключовата дума **this** преди типа и името на първия аргумент на метода (в случая **str**). Самият метод е статичен и е дефиниран в статичния клас **StringExtensions**. Използването на разширяващия метод става както всеки обикновен метод на класа **string**. Не забравяйте да добавите съответния namespace, в който се намира статичния клас, описващ разширяващите методи.

Пример:

```
static void Main()
{
   string helloString = "Hello, Extension Methods!";
   int wordCount = helloString.WordCount();
   Console.WriteLine(wordCount);
}
```

Самият метод се вика върху обекта **helloString**, който е от тип **string**. Методът получава обекта като аргумент и работи с него (в случая вика неговия метод **Split(…)** и връща броя елементи в получения масив).

Разширяващи методи за интерфейси

Освен върху класове, разширяемите методи могат да работят и върху интерфейси. Следващият пример взима обект от клас, който имплементира интерфейса списък от цели числа (**IList<int>**) и увеличава тяхната стойност с определено число. Самия метод **IncreaseWidth(…)** има достъп само до елементите, които се включват в интерфейса **IList** (например свойството **Count**).

```
public static class IListExtensions
{
   public static void IncreaseWidth(
      this IList<int> list,
      int amount)
   {
      for (int i = 0; i < list.Count; i++)
      {
            list[i] += amount;
        }
    }
}
```

Разширяващите методи предоставят и възможност за работа върху generic типове. Нека вземем за пример метод, който обхожда с оператора **`foreach`** дадена колекция, имплементираща **`IEnumerable`** от произволен тип **`T`**:

```
public static class IEnumerableExtensions
{
    public static string ToString<T>(
        this IEnumerable<T> enumeration)
    {
        StringBuilder result = new StringBuilder();
        result.Append("[");
        foreach (var item in enumeration)
        {
            result.Append(item.ToString());
            result.Append(", ");
        }
        if (result.Length > 1)
            result.Remove(result.Length - 2, 2);
        result.Append("]");
        return result.ToString();
    }
}
```

Пример за употребата на горните два метода:

```
static void Main()
{
    List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
    Console.WriteLine(numbers.ToString<int>());
    numbers.IncreaseWidth(5);
    Console.WriteLine(numbers.ToString<int>());
}
```

Резултатът от изпълнението на програмата ще е следният:

```
[1, 2, 3, 4, 5]
[6, 7, 8, 9, 10]
```

Анонимни типове (anonymous types)

В обектно-ориентираните езици (какъвто е C#) много често се налага да се дефинират малки класове с цел еднократно използване. Типичен пример за това е класа **`Point`**, съдържащ само 2 полета – координатите на точка. Създаването на обикновен клас само и единствено за еднократна употреба създава неудобство на програмистите и е свързано със загуба на време, особено за предефиниране на стандартните за всеки клас операции **`ToString()`**,**`Equals()`** и **`GetHashCode()`**.

В езика C# има вграден начин за създаване на типове за еднократна употреба, наричани **анонимни типове (anonymous types)**. Обектите от такъв тип се създават почти по същия начин, по който се създават стандартно обектите в C#. При тях не е нужно предварително да дефинираме тип данни за променливата. С ключовата дума **`var`** показваме на компилатора, че типа на променливата трябва да се разбере автоматично от дясната страна на присвояването. Реално нямаме и друг избор, тъй като дефинираме променлива от анонимен тип, на която не можем да посочим конкретно от кой тип е. След това пишем името на обекта, оператора равно и ключовата дума **`new`**. Във фигурни скоби изреждаме имената на свойствата на анонимния тип и техните стойности.

Анонимни типове – пример

Ето един пример за създаване на анонимен тип, който описва лек автомобил:

```
var myCar = new { Color = "Red", Brand = "BMW", Speed = 180 };
```

По време на компилация, компилаторът ще създаде клас с уникално име (например `<>f__AnonymousType0`), ще му създаде свойства (с getter и setter). В горния пример за свойствата **`Color`** и **`Brand`** компилаторът сам ще се досети, че са от тип **`string`**, а за свойството **`Speed`**, че е от тип **`int`**. Веднага след инициализацията си, обектът от анонимния тип може да бъде използван като обикновен тип с трите си свойства:

```
Console.WriteLine("My car is a {0} {1}.",
  myCar.Color, myCar.Brand);
Console.WriteLine("It runs {0} km/h.", myCar.Speed);
```

Резултатът от изпълнението на горния код ще е следния:

```
My car is a Red BMW.
It runs 180 km/h.
```

Още за анонимните типове

Както всеки друг тип в .NET и анонимните типове наследяват **`System.Object`**. По време на компилация, компилаторът ще предефинира вместо нас методите **`ToString()`**, **`Equals()`** и **`GetHashCode()`**.

```
Console.WriteLine("ToString: {0}", myCar.ToString());
Console.WriteLine("Hash code: {0}",
  myCar.GetHashCode().ToString());
Console.WriteLine("Equals? {0}", myCar.Equals(
  new { Color = "Red", Brand = "BMW", Speed = 180 }
));
Console.WriteLine("Type name: {0}", myCar.GetType().ToString());
```

Резултатът от изпълнението на горния код ще е следния:

```
ToString: { Color = Red, Brand = BMW, Speed = 180 }
Hash code: 1572002086
Equals? True
Type name:
<>f__AnonymousType0`3[System.String,System.String,System.Int32]
```

Както може да се види от резултата, методът **`ToString()`** е предефиниран така, че да изрежда свойствата на анонимния тип в реда, в който сме ги дефинирали при инициализацията на обекта (в случая **`myCar`**). Методът **`GetHashCode()`** е реализиран така, че да взима предвид всички полета и спрямо тях да изчислява собствена хеш-функция с малък брой колизии. Предефинираният от компилатора метод **`Equals(…)`** сравнява по стойност обектите. Както може да се види от примера, създаваме нов обект, който има абсолютно същите свойства като **`myCar`** и получаваме като резултат от метода, че новосъздаденият обект и старият са еднакви по стойност.

Анонимните типове, както и обикновените, могат да бъдат елементи на масиви. Инициализирането отново става с ключовата дума **`new`** като след нея се слагат квадратни скоби. Стойностите на масива се изреждат по начина, по който се задават стойности на анонимни типове. Стойностите в масива трябва да са хомогенни, т.е. не може да има различни анонимни типове в един и същ масив. Пример за дефиниране на масив от анонимни типове с 2 свойства (**`X`** и **`Y`**):

```
var arr = new[]  {  new { X = 3, Y = 5 },
                    new { X = 1, Y = 2 },
                    new { X = 0, Y = 7 }
                 };
foreach (var item in arr)
{
  Console.WriteLine(item.ToString());
}
```

Резултатът от изпълнението на горния код ще е следния:

```
{ X = 3, Y = 5 }
{ X = 1, Y = 2 }
{ X = 0, Y = 7 }
```

Ламбда изрази (lambda expressions)

Ламбда изразите представляват анонимни функции, които съдържат изрази или последователност от оператори. Всички ламбда изрази използват ламбда оператора **=>**, който може да се чете като "отива в". Идеята за ламбда изразите в C# е взаимствана от функционалните езици (например **Haskell**, **Lisp**, **Scheme**, **F#** и др.). Лявата страна на ламбда оператора определя входните параметри на анонимната функция, а дясната страна представлява израз или последователност от оператори, която работи с входните параметри и евентуално връща някакъв резултат.

Обикновено ламбда изразите се използват като предикати или вместо делегати (променливи от тип функция), които се прилагат върху колекции, обработвайки елементите от колекцията по някакъв начин и/или връщайки определен резултат.

Ламбда изрази – примери

Например нека да разгледаме разширяващия метод **FindAll(…)**, който може да се използва за отсяване на необходимите елементи. Той работи върху определена колекция, прилагайки ѝ даден предикат, който проверява всеки от елементите на колекцията дали отговаря на определено условие. За да го използваме, обаче, трябва да включим референция към библиотеката **System.Core.dll** и namespace-а **System.Linq**, тъй като разширяващите методи върху колекциите се намират в този namespace.

Ако искаме например да вземем само четните числа от колекция с цели числа, можем да използваме метода **FindAll(…)** върху колекцията, като му подадем ламбда метод, който да провери дали дадено число е четно:

```
List<int> list = new List<int>() { 1, 2, 3, 4, 5, 6 };
List<int> evenNumbers = list.FindAll(x => (x % 2) == 0);
foreach (var num in evenNumbers)
{
  Console.Write("{0} ", num);
}
Console.WriteLine();
```

Резултатът е:

```
2 4 6
```

Горният пример обхожда цялата колекция от числа и за всеки елемент от нея (именуван **x**) се прави проверка дали числото се дели на 2 (с булевия израз **(x % 2) == 0**).

Нека сега разгледаме един пример, в който чрез разширяващ метод и ламбда израз ще създадем колекция, съдържаща определена информация от даден клас. В случая от класа куче - **`Dog`** (със свойства име **`Name`** и възраст **`Age`**), искаме да получим списък само с имената на кучетата. Това можем да направим с разширяващия метод **`Select(…)`**(дефиниран в namespace `System.Linq`), като му зададем за всяко куче **`x`** да го превръща в името на кучето (**`x.Name`**) и върнатия резултат (колекция) да запише в променливата **`names`**. С ключовата дума **`var`** казваме на компилатора сам да се определи типа на променливата по резултата, който присвояваме в дясната страна.

```
class Dog
{
  public string Name { get; set; }
  public int Age { get; set; }
}

static void Main()
{
  List<Dog> dogs = new List<Dog>()
  {
    new Dog { Name = "Rex", Age = 4 },
    new Dog { Name = "Sharo", Age = 0 },
    new Dog { Name = "Stasi", Age = 3 }
  };
  var names = dogs.Select(x => x.Name);
  foreach (var name in names)
  {
    Console.WriteLine(name);
  }
}
```

Резултатът е:

```
Rex
Sharo
Stasi
```

Използване на ламбда изрази с анонимни типове

С ламбда изрази можем да създаваме и колекции с анонимни типове, от колекция с някакви елементи. Например нека от колекцията **`dogs`**, съдържаща елементи от тип **`Dog`** да създадем нова колекция с елементи от анонимен тип, с 2 свойства – възраст и първата буква от името на кучето:

```
var newDogsList = dogs.Select(
  x => new { Age = x.Age, FirstLetter = x.Name[0] });
foreach (var item in newDogsList)
{
   Console.WriteLine(item);
}
```

Резултатът е:

```
{ Age = 4, FirstLetter = R }
{ Age = 0, FirstLetter = S }
{ Age = 3, FirstLetter = S }
```

Както може да се види от примера, новосъздадената колекция **newDogsList** е с елементи от анонимен тип, съдържащ свойствата **Age** и **FirstLetter**. Първият ред от примера може да се прочете така: *създай ми променлива с неизвестен за сега тип, именувай я* ***newDogsList*** *и от* ***dogs*** *колекцията, за всеки неин елемент* ***x*** *създай нов анонимен тип с 2 свойства:* ***Age****, което е равно на свойството* ***Age*** *от елемента* ***x*** *и свойство* ***FirstLetter****, което пък е равно на първия символ от низа* ***x.Name***.

Сортиране чрез ламбда изрази

Ако искаме да сортираме елементите в дадена колекция, можем да използваме разширяващите методи **OrderBy** и **OrderByDescending** като им подадем чрез ламбда функция начина, по който да сортират елементите. Пример отново върху колекцията **dogs**:

```
var sortedDogs = dogs.OrderByDescending(x => x.Age);
foreach (var dog in sortedDogs)
{
   Console.WriteLine(string.Format(
      "Dog {0} is {1} years old.", dog.Name, dog.Age));
}
```

Резултатът е:

```
Dog Rex is 4 years old.
Dog Stasi is 3 years old.
Dog Sharo is 0 years old.
```

Оператори в ламбда изразите

Ламбда функциите могат да имат и тяло. До сега използвахме ламбда функциите само с един оператор. Сега ще разгледаме ламбда функции, които имат тяло. Да се върнем на примера с четните числа. За всяко число, към което се прилага нашата ламбда функция, искаме да отпечатаме на конзолата стойността му и да върнем като резултат дали е четно или не:

```
List<int> list = new List<int>() { 20, 1, 4, 8, 9, 44 };
// Process each argument with code statements
var evenNumbers = list.FindAll((i) =>
{
   Console.WriteLine("Value of i is: {0}", i);
   return (i % 2) == 0;
});
```

```
Value of i is: 20
Value of i is: 1
Value of i is: 4
Value of i is: 8
Value of i is: 9
Value of i is: 44
```

Ламбда изразите като делегати

Ламбда функциите могат да бъдат записани в променливи от тип делегат. Делегатите представляват специален тип променливи, които съдържат функции. Стандартните типове делегати в .NET са **Action**, **Action<in T>**, **Action<in T1, in T2>**, и т.н. и **Func<out TResult>**, **Func<in T, out TResult>**, **Func<in T1, in T2, in TResult>** и т.н. Типовете **Func** и **Action** са generic и съдържат типовете на връщаната стойност и типовете на параметрите на функциите. Променливите от тези типове са референции към функции. Ето пример за използването и присвояването на стойности на тези типове.

```
Func<bool> boolFunc = () => true;
Func<int, bool> intFunc = (x) => x < 10;
if (boolFunc() && intFunc(5))
{
   Console.WriteLine("5 < 10");
}
```

Резултатът е:

```
5 < 10
```

В горния пример дефинираме два делегата. Първият делегат **boolFunc** е функция, която няма входни параметри и връща като резултат от булев тип. На нея като стойност сме задали анонимна ламбда функция която не върши нищо и винаги връща стойност **true**. Вторият делегат приема като параметър променлива от тип **int** и връща булева стойност, която е истина, когато входния параметър **x** е по-малък от 10 и лъжа в противен случай. Накрая в **if** оператора викаме нашите два делегата, като на втория даваме параметър 5 и резултата от извикването им, както може да се види и на двата е **true**.

LINQ заявки (LINQ queries)

LINQ (Language-Integrated Query) представлява редица разширения на .NET Framework, които включват интегрирани в езика заявки и операции върху елементи от даден източник на данни (най-често масиви и колекции). LINQ е много мощен инструмент, който доста прилича на повечето SQL езици и по синтаксис и по логика на изпълнение. LINQ реално обработва колекциите по подобие на SQL езиците, които обработват редовете в таблици в база данни. Той е част от **C#** и **VisualBasic** синтаксиса и се състои от няколко основни ключови думи. За да използваме LINQ заявки в езика C#, трябва да включим референция към **System.Core.dll** и да добавим namespace-а **System.Linq**.

Избор на източник на данни с LINQ

С ключовите думи **from** и **in** се задават източникът на данни (колекция, масив и т.н.) и променливата, с която ще се итерира (обхожда) по колекцията (обхождане по подобие на **foreach** оператора). Например заявка, която започва така:

```
from culture
in CultureInfo.GetCultures(CultureTypes.AllCultures)
```

може да се прочете като: *За всяка една стойност от колекцията CultureInfo.GetCultures(CultureTypes.AllCultures) задай име culture, и го използвай по-нататък в заявката…*

Филтриране на данните с LINQ

С ключовата дума **where** се задават условията, които всеки от елементите от колекцията трябва да изпълнява, за да продължи да се изпълнява заявката за него. Изразът след **where** винаги е булев израз. Може да се каже, че с **where** се филтрират елементите. Например, ако искаме в предния пример да кажем, че ни трябват само тези от културите, чието име започва с малка латинска буква **b**, можем да продължим заявката с:

```
where culture.Name.StartsWith("b")
```

Както може да се забележи, след **where…in** конструкцията използваме само името, което сме задали за обхождане на всяка една променлива от колекцията. Ключовата дума **where** се компилира до извикване на extension метода **Where()**.

Избор на резултат от LINQ заявката

С ключовата дума **select** се задават какви данни да се върнат от заявката. Резултата от заявката е под формата на обект от съществуващ клас или анонимен тип. Върнатият резултат може да бъде и свойство на обектите, които заявката обхожда или самите обекти. Операторът **`select`** и всичко след него седи винаги в края на заявката. Ключовите думи **`from`**, **`in`**, **`where`** и **`select`** са достатъчни за създаването на проста LINQ заявка. Ето и пример:

```
List<int> numbers = new List<int>() {
   1, 2, 3, 4, 5, 6, 7, 8, 9, 10
};
var evenNumbers =
   from num in numbers
   where num % 2 == 0
   select num;
foreach (var item in evenNumbers)
{
   Console.Write(item + " ");
}
```

Резултатът е:

```
2 4 6 8 10
```

Горния пример прави заявка върху колекцията от числа **`numbers`** и записва в нова колекция само четните числа. Заявката може да се прочете така: *За всяко число **`num`** от **`numbers`** провери дали се дели на 2 без остатък и ако е така го добави в новата колекция.*

Сортиране на данните с LINQ

Сортирането чрез LINQ заявките се извършва с ключовата дума **`orderby`**. След нея се слагат условията, по които да се подреждат елементите, участващи в заявката. За всяко условие може да се укаже редът на подреждане: в нарастващ ред (с ключова дума **`ascending`**) или в намаляващ ред (с ключова дума **`descending`**) като по подразбиране се подреждат в нарастващ ред. Например, ако искаме да сортираме масив от думи по дължината им в намаляващ ред, можем да напишем следната LINQ заявка:

```
string[] words = { "cherry", "apple", "blueberry" };
var wordsSortedByLength =
   from word in words
   orderby word.Length descending
   select word;
foreach (var word in wordsSortedByLength)
{
   Console.WriteLine(word);
}
```

Резултатът е:

```
blueberry
cherry
apple
```

Ако не бъде указан начина, по който да се сортират елементите (т.е. ключовата дума **orderby** отсъства от заявката), се взимат елементите в реда, в който колекцията би ги върнала при обхождане с **foreach** оператора.

Групиране на резултатите с LINQ

С ключовата дума **group** се извършва групиране на резултатите по даден критерии. Форматът е следният:

group [име на променливата] **by** [признак за групиране] **into** [име на групата]

Резултатът от това групиране е нова колекция от специален тип, която може да бъде използвана по-надолу в заявката. След групирането, обаче, заявката спира да работи с първоначалната си променлива. Това означава, че в **select**-а може да се ползва само групата. Пример за групиране:

```
int[] numbers =
  { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0, 10, 11, 12, 13 };
int dividor = 5;

var numberGroups =
  from number in numbers
  group number by number % divisor into group
  select new { Remainder = group.Key, Numbers = group };

foreach (var group in numberGroups)
{
  Console.WriteLine(
  "Numbers with a remainder of {0} when divided by {1}:",
    group.Remainder, divisor);
  foreach (var number in group.Numbers)
  {
    Console.WriteLine(number);
  }
}
```

Резултатът е:

```
Numbers with a remainder of 0 when divided by 5:
5
```

```
0
10
Numbers with a remainder of 4 when divided by 5:
4
9
Numbers with a remainder of 1 when divided by 5:
1
6
11
Numbers with a remainder of 3 when divided by 5:
3
8
13
Numbers with a remainder of 2 when divided by 5:
7
2
12
```

Както може да се види от примера, на конзолата се извеждат числата, групирани по остатъка си от деление с 5. В заявката за всяко число се смята number % divisor и за всеки различен резултат се прави нова група. По-надолу **`select`** оператора работи върху списъка от създадените групи и за всяка група създава анонимен тип, който съдържа 2 свойства: **`Remainder`** и **`Numbers`**. На свойството **`Remainder`** се присвоява ключа на групата (в случая остатъка от деление с **`divisor`** на числото). На свойството **`Numbers`** пък се присвоява колекцията group, която съдържа всички елементи в групата. Забележете, че **`select`**-а се изпълнява само и единствено върху списъка от групи. Там не може да се използва променливата **`number`**. По-натам в примера с 2 вложени **`foreach`** оператора се извеждат остатъците (групите) и числата, които имат остатъка (се намират в групата).

Съединение на данни с LINQ

Операторът **`join`** има доста по-сложна концепция от останалите LINQ оператори. Той съединява колекции по даден критерии (еднаквост) между тях и извлича необходимата информация от тях. Синтаксисът му е следният:

`from` [име на променлива от колекция 1] **`in`** [колекция 1]
`join` [име на променлива от колекция 2] **`in`** [колекция 2] **`on`** [част на условието за еднаквост от колекция 1] **`equals`** [част на условието за еднаквост от колекция 2]

По-надолу в заявката (в **`select`**-а например) може да се използва, както името на променливата от колекция 1, така и това от колекция 2. Пример:

```
public class Product
{
  public string Name { get; set; }
  public int CategoryID { get; set; }
}
public class Category
{
  public int ID { get; set; }
  public string Name { get; set; }
}
```

Резултатът е:

```
List<Category> categories = new List<Category>()
{
  new Category() { ID = 1, Name = "Fruit" },
  new Category() { ID = 2, Name = "Food" },
  new Category() { ID = 3, Name = "Shoe" },
  new Category() { ID = 4, Name = "Juice" },
};
List<Product> products = new List<Product>()
{
  new Product() { Name = "Strawberry", CategoryID = 1 },
  new Product() { Name = "Banana", CategoryID = 1 },
  new Product() { Name = "Chicken meat", CategoryID = 2 },
  new Product() { Name = "Apple Juice", CategoryID = 4 },
  new Product() { Name = "Fish", CategoryID = 2 },
  new Product() { Name = "Orange Juice", CategoryID = 4 },
  new Product() { Name = "Sandal", CategoryID = 3 },
};
var productsWithCategories =
  from product in products
  join category in categories
    on product.CategoryID equals category.ID
  select new {  Name = product.Name,
                Category = category.Name };
foreach (var item in productsWithCategories)
{
  Console.WriteLine(item);
}
```

Резултатът е:

```
{ Name = Strawberry, Category = Fruit }
{ Name = Banana, Category = Fruit }
{ Name = Chicken meat, Category = Food }
{ Name = Apple Juice, Category = Juice }
{ Name = Fish, Category = Food }
{ Name = Orange Juice, Category = Juice }
{ Name = Sandal, Category = Shoe }
```

В горния пример си създаваме два класа и мислена релация (връзка) между тях. На всеки продукт съответства някаква категория **`CategoryID`** (под формата на число), което отговаря на числото **`ID`** от класа **`Category`** в колекцията categories. Ако искаме да използваме тази релация и да си създадем нов анонимен тип, в който да запишем продуктите с тяхното име и името на тяхната категория, си пишем горната LINQ заявка. Тя свързва колекцията елементи от тип **`Category`** с колекцията от елементи от тип **`Product`** по споменатия признак (еднаквост между **`ID`** от **`Category`** и **`CategoryID`** от **`Products`**). В select частта на заявката използваме двете имена category и product, за да си конструираме анонимен тип с име на продукта и име на категорията.

Вложени LINQ заявки

В LINQ се поддържат и вложени заявки. Например последната заявка може да бъде написана чрез влагането на заявка в заявка по следния начин, като резултата е абсолютно същия както в заявката с **`join`**:

```
var productsWithCategories =
   from product in products
   select new {
      Name = product.Name,
      Category =
         (from category in categories
         where category.ID == product.CategoryID
         select category.Name).First()
   };
```

Тъй като всяка LINQ заявка връща колекция от елементи (без значение дали резултатът от нея е с 0, 1 или няколко елемента), се налага използването на разширяващия метод **`First()`** върху резултата от вложената заявка. Методът **`First()`** връща първия елемент (в нашия случай и единствен) от колекцията, върху която е приложен. По този начин получаваме името на категорията само по нейния **`ID`** номер.

Упражнения

1. Имплементирайте разширяващ метод **`Substring(int index, int length)`** за класа **`StringBuilder`**, който връща нов **`StringBuilder`** и има същата функционалност като метода **`Substring(…)`** на класа **`String`**.

2. Имплементирайте следните разширяващи методи за класовете, имплементиращи интерфейса **`IEnumerable<T>`**: **`Sum`**, **`Min`**, **`Max`**, **`Average`**.

3. Напишете клас **`Student`** със следните свойства: първо име, фамилия и възраст. Напишете метод, който по даден масив от студенти намира всички студенти, на които името им е по-малко лексикографски от фамилията. Използвайте LINQ заявка.
4. Напишете LINQ заявка, която намира първото име и фамилията на всички студенти, които са на възраст между 18 и 24 години включително. Използвайте класа **`Student`** от предната задача.
5. Като използвате разширяващите методи **`OrderBy(…)`** и **`ThenBy(…)`** с ламбда израз, сортирайте списък от студенти по първо име и по фамилия в намаляващ лексикографски ред. Напишете същата функционалност, използвайки LINQ заявка.
6. Напишете програма, която отпечатва на конзолата всички числа в даден масив (или списък), които се делят едновременно на 7 и на 3. Използвайте вградените разширяващи методи с ламбда изрази и после напишете същото, само че с LINQ заявка.
7. Напишете разширяващ метод на класа **`String`**, който прави главна, всяка буква, която е начало на дума в изречение на английски език. Например текстът "**`this iS a Sample sentence.`**" трябва да стане на "**`This Is A Sample Sentence.`**".

Решения и упътвания

1. Едно решение на задачата е да направите нов **`StringBuilder`** и в него да запишете символите с индекси започващи от **`index`** и с дължина **`length`** от обекта върху който ще работи разширяващият метод.
2. Тъй като не всички класове имат предефинирани операторите **`+`** и **`/`**, операциите **`Sum`** и **`Average`** няма да могат да бъдат приложени директно върху тях. Един начин за справяне с този проблем е да конвертираме всеки обект към обект от тип **`decimal`** и после да извършим операциите върху тях. За конвертирането може да се използва статичният метод **`Convert.ToDecimal(…)`**. За операциите **`Min`** и **`Max`** може да се зададе на темплейтния клас да наследява винаги **`IComparable`**, за да могат обектите да бъдат сравнявани.
3. Прегледайте ключовите думи **`from`**, **`where`** и **`select`** от секцията LINQ заявки.
4. Използвайте LINQ заявка, за да създадете анонимен тип, който съдържа само 2 свойства – **`FirstName`** и **`LastName`**.
5. За LINQ заявката използвайте **`from`**, **`orderby`**, **`descending`** и **`select`**. За реализацията с ламбда изразите използвайте функциите **`OrderByDescending(…)`** и **`ThenByDescending(…)`**.

6. Вместо да правите 2 условия за **`where`** е достатъчно само да проверите дали числата се делят на 21.

7. Използвайте метода **`ToLittleCase(…)`** на свойството **`TextInfo`** в културата **`en-US`** по следния начин:

```
new CultureInfo("en-US", false).TextInfo.ToTitleCase(text);
```

Глава 23. Как да решаваме задачи по програмиране?

В тази тема...

В настоящата тема ще дискутираме един препоръчителен подход за решаване на задачи по програмиране и ще го илюстрираме нагледно с реални примери. Ще дискутираме инженерните принципи, които трябва да следваме при решаването на задачи по програмиране (които важат в голяма степен и за задачи по математика, физика и други дисциплини) и ще ги покажем в действие. Ще опишем стъпките, през които преминаваме при решаването на няколко примерни задачи и ще демонстрираме какви грешки се получават, ако не следваме тези стъпки. Ще обърнем внимание на някои важни стъпки от решаването на задачи, които обикновено се пропускат, като например тестването. Надяваме се да успеем да ви докажем чрез много примери, че за решаването на задачи по програмиране си има "рецепта" и да ви убедим колко много помага тя.

Основни принципи при решаване на задачи по програмиране

Сигурно си мислите, че сега ще ви напълним главата с празни приказки в стил "първо мисли, след това пиши" или "внимавайте като пишете, че да не пропуснете нещо". Всъщност тази тема няма да е толкова досадна и ще ви даде практически насоки как да подхождате при решаването на задачи, независимо дали са алгоритмични или други.

Без да претендираме за изчерпателност, ще ви дадем няколко важни препоръки, базирани на опита на Светлин Наков, който повече от 10 години подред е участвал редовно по български и международни състезания по програмиране, а след това е обучавал на програмиране и решаване на задачи студенти в Софийски университет "Св. Климент Охридски" (ФМИ на СУ), в Нов Български Университет (НБУ), в Национална академия по разработка на софтуер (НАРС), както и в Telerik Academy.

Нека започнем с първата важна препоръка.

Използвайте лист и химикал!

Захващането на лист и химикал и скицирането на примери и разсъждения по дадения проблем е нещо съвсем нормално и естествено – нещо, което всеки опитен математик, физик или софтуерен инженер прави, когато му поставят нетривиална задача.

За съжаление, от опита си с обучението на софтуерни инженери в НАРС можем да споделим, че повечето начинаещи програмисти въобще не си носят лист и химикал. Те имат погрешното съзнание, че за да решават задачи по програмиране им е достатъчна само клавиатурата. На повечето им трябват доста време и провали по изпитите, за да достигат до важния извод, че използването на някаква форма на чертеж, скица или визуализация на проблема е от решаваща полза за неговото разбиране и за конструиране на правилно решение.

Който не ползва лист и химикал, ще бъде силно затруднен при решаването на задачи по програмиране. Винаги скицирайте идеите си на хартия или на дъската преди да започнете да пишете на клавиатурата!

Наистина изглежда старомодно, но ерата на хартията все още не е отминала! Най-лесният начин човек да си скицира идеите и разсъжденията е като хване лист и химикал, а без да скицирате идеите си, е много трудно да разсъждавате. Чисто психологически това е свързано с визуалната система за представяне на информацията в човешкия мозък, която работи изключително бързо и е свързана силно с творческия потенциал и с логическото мислене. Хората с развита визуална система първо си представят решението и го "виждат" по някакъв начин в своето съзнание, а след това го развиват като идея и накрая стигат до реализация. Те използват активно визуалната си памет и способността си визуално да конструират образи, което им дава възможност много бързо да разсъждават. Такива хора за секунди могат да прехвърлят през съзнанието си десетки идеи и да си представят алгоритмите и решенията на задачите. Независимо дали сте от "визуалния" тип хора или не, да си скицирате проблема или да си го нарисувате ще ви помогне на разсъжденията, с които да достигнете до решението му, защото всеки има способност да си представя нещата визуално.

Помислете например колко усилия ви трябват, за да умножавате петцифрени числа на ум и колко по-малко са усилията, ако имате лист и химикал (изключваме възможността да използваме електронни устройства). По същия начин е с решаването на задачи – когато трябва да измислите решение, ви трябва лист хартия за да си драскате и рисувате. Когато трябва да проверите дали решението ви е вярно, ви трябва отново хартия, за да си разпишете един пример. Когато трябва да измисляте случаи, които вашето решение изпуска, отново ви трябва нещо, на което да си разписвате и драскате примери и идеи. Затова ползвайте лист и химикал!

Генерирайте идеи и ги пробвайте!

Решаването на дадена задача винаги започва от скицирането на някакъв пример върху лист хартия. Когато имате конкретен пример, можете да разсъждавате, а когато разсъждавате, ви хрумват идеи за решение на задачата.

Когато вече имате идея, ви трябват още примери, за да проверите дали идеята е добра. Тогава можете да нарисувате още няколко примера на хартия и да пробвате вашата идея върху тях. Уверете се, че идеята ви е вярна. Проследете идеята стъпка по стъпка, така, както ще я изпълни евентуална компютърна програма и вижте дали няма някакви проблеми.

Опитайте се да "счупите" вашата идея за решение – да измислите пример, при който тя не работи (контра-пример). Ако не успеете, вероятно сте на прав път. Ако успеете, помислете как да се справите с неработещия пример: измислете "поправка" на вашата идея за алгоритъм или измислете напълно нова идея.

Не винаги първата идея, която ви хрумва, е правилна и може да се превърне в решение на задачата. Решаването на задачи е итеративен процес, при който последователно измисляте идеи и ги пробвате върху различни примери докато не стигнете до идея, която изглежда, че е правилна и може успешно да реши задачата. Понякога могат да минат часове в опитите ви да измислите алгоритъм за решаването на дадена задача и да пробвате десетки различни идеи. Това е нормално. Никой няма способността да измисля моментално решение на всяка задача, но със сигурност колкото по-голям опит имате при решаването на задачи, толкова по-бързо ще ви идват добри идеи. Ако сте решавали подобна задача, бързо ще се сетите за нея и за начина по който сте я решили, тъй като едно от основните свойства на човешкия мозък е да разсъждава с аналогии. Опитът от решаването на даден тип задачи ви научава бързо да измисляте решение по аналогия с друга подобна задача.

За да измисляте идеи и да ги проверявате ви трябват лист, химикал и различни примери, които да измисляте и да визуализирате чрез скица, рисунка, чертеж или друг способ. Това ви помага много бързо да пробвате различни идеи и да разсъждавате върху идеите, които ви хрумват. Основното действие при решаването на задачи е да разсъждавате логически, да търсите аналогии с други задачи и методи, да обобщавате или да прилагате обобщени идеи и да конструирате решението си като си го визуализирате на хартия. Когато имате скица или чертеж вие можете да си представяте визуално какво би се случило, ако извършим дадено действие върху данните от картинката. Това може да ни даде и идея за следващо действие или да ни откаже от нея. Така може да стигнем до цялостен алгоритъм, чиято коректност можем да проверим като го разпишем върху конкретен пример.

Решаването на задачи по програмиране започва от измислянето на идеи и проверяването им. Това става най-лесно като хванете лист и химикал и скицирате разсъжденията си. Винаги проверявайте идеите си с подходящи примери!

Горните препоръки са много полезни и в още един случай: когато сте на интервю за работа. Всеки опитен интервюиращ може да потвърди, че когато даде алгоритмична задача на кандидат за работа, очаква от него да хване лист и химикал и да разсъждава на глас като предлага различни идеи, които му хрумват. Хващането на лист и химикал на интервю за работа дава признаци за мислене и правилен подход за решаване на проблеми. Разсъждаването на глас показва, че можете да мислите. Дори и да не стигнете до правилно решение подходът към решаване на задачи ще направи добро впечатление на интервюиращия!

Разбивайте задачата на подзадачи!

Сложните задачи винаги могат да се разделят на няколко по-прости. Ще ви покажем това в примерите след малко. Нищо сложно на този свят не е направено наведнъж. Рецептата за решаване на сложни задачи е да се разбият логически на няколко по-прости (по възможност максимално независими една от друга). Ако и те се окажат сложни, разбиването на по-прости може да се приложи и за тях. Тази техника е известна като "разделяй и владей" и е използвана още в Римската империя.

Разделянето на проблема на части звучи просто на теория, но на практика не винаги е лесно да се направи. Тънкостта на решаване на алгоритмични задачи се крие в това да овладеете добре техниката на разбиването на задачата на по-прости подзадачи и, разбира се, да се научите да ви хрумват добри идеи, което става с много, много практика.

Сложните проблеми винаги могат да се разделят на няколко по-прости. Когато решавате задачи, разделяйте сложната задача на няколко по-прости задачи, които можете да решите самостоятелно.

Разбъркване на тесте карти – пример

Нека дадем един пример: трябва да разбъркаме тесте карти в случаен ред. Да приемем, че тестето е дадено като масив или списък от N на брой обекти (всяка карта е обект). Това е задача, която изисква много стъпки (някаква серия от изваждания, вмъквания, размествания или преподреждания на карти). Тези стъпки сами по себе си са по-прости и по-лесни за реализация, отколкото цялостната задача за разбъркване на картите. Ако намерим начин да разбием сложната задача на множество простички стъпки, значи сме намерили начин да я решим. Именно в това се състои алгоритмичното мислене: в умението да разбиваме сложен проблем на серия по-прости проблеми, за които можем да намерим решение. Това, разбира се, важи не само за програмирането, но и за решаването на задачи по математика, физика и други дисциплини. Точно алгоритмичното мислене е причината математиците и физиците много бързо да напредват, когато се захванат с програмиране.

Нека се върнем към нашата задача и да помислим кои са елементарните действия, които са нужни, за да разбъркаме в случаен ред картите?

Ако хванем в ръка тесте карти или си го нарисуваме по някакъв начин на лист хартия (например като серия кутийки с по една карта във всяка от тях), веднага ще ни хрумне идеята, че е необходимо да направим някакви размествания или пренареждания на някои от картите.

Разсъждавайки в този дух стигаме до заключението, че трябва да направим повече от едно разместване на една или повече карти. Ако направим само едно разместване, получената подредба няма да е съвсем случайна. Следователно ни трябват много на брой по-прости операции за единични размествания.

Стигнахме до първото разделяне на задачата на подзадачи: трябват ни серия размествания и всяко разместване можем да разгледаме като по-проста задача, част от решението на по-сложната.

Първа подзадача: единично разместване

Как правим "единично разместване" на карти в тестето? На този въпрос има стотици отговори, но можем да вземем първата идея, която ни хрумва. Ако е добра, ще я ползваме. Ако не е добра, ще измислим друга.

Ето каква може да е първата ни идея: ако имаме тесте карти, можем да се сетим да разделим тестето на две части по случаен начин и да разменим едната част с другата. Имаме ли идея за "единично разместване" на картите? Имаме. Остава да видим дали тази идея ще ни свърши работа (ще я пробваме след малко на практика).

Нека се върнем на началната задача: трябва да получим случайно размесено тестето карти, което ни е дадено като вход. Ако хванем тестето и много на брой пъти го разцепим на две и разменим получените две части, ще получим случайно размесване, нали? Изглежда нашата първа идея за "единично разместване" ще свърши работа.

Втора подзадача: избор на случайно число

Как избираме случаен начин за разцепване на тестето? Ако имаме N карти, ни трябва начин да изберем число между 1 и N-1, нали?

За да решим тази подзадача, ни трябва или външна помощ, или да знаем, че тази задача в .NET Framework е вече решена и можем да ползваме вградения генератор на случайни числа наготово.

Ако не се сетим да потърсим в Интернет как със C# се генерират случайни числа, можем да си измислим и наше собствено решение, например да въвеждаме един ред от клавиатурата и да измерваме интервала време между стартирането на програмата и натискането на [Enter] за край на въвеждането. Понеже при всяко въвеждане това време ще е различно (особено, ако можем да отчитаме с точност до наносекунди), ще имаме начин да получим случайно число. Остава въпросът как да го накараме да бъде в интервала от 1 до N-1, но вероятно ще се сетим да ползваме остатъка от деление на (N-1) и да си решим проблема.

Виждате, че дори простите задачи могат да имат свои подзадачи или може да се окаже, че за тях вече имаме готово решение. Когато намерим решение, приключваме с текущата подзадача и се връщаме към оригиналната задача, за да продължим да търсим идеи и за нейното решаване. Нека направим това.

Трета подзадача: комбиниране на разместванията

Да се върнем пак на началната задача. Чрез последователни разсъждения стигнахме до идеята много пъти да извършим операцията "единично разместване" в тестето карти докато тестето се размести добре. Това изглежда коректно и можем да го пробваме.

Сега възниква въпросът колко пъти да извършим операцията "единично разместване". 100 пъти достатъчно ли е? А не е ли много? А 5 пъти достатъчно ли е, не е ли малко? За да дадем добър отговор на този въпрос трябва да помислим малко. Колко карти имаме? Ако картите са малко, ще ни трябват малко размествания. Ако картите са много, ще ни трябват повече размествания, нали? Следователно броят размествания изглежда зависи от броя карти.

За да видим колко точно трябва да са тези разместваниия, можем да вземем за пример стандартно тесте карти. Колко карти има в него? Всеки картоиграч ще каже, че са 52. Ами тогава да помислим колко разцепвания на тестето на две и разменяния на двете половини ни трябват, за да разбъркаме случайно 52 карти. Дали 52 е добре? Ако направим 52 "единични размествания" изглежда, че ще е достатъчно, защото заради случайния избор ще сцепим средно по 1 път между всеки две карти (това е видно и без да четем дебели книги по вероятности и статистика). А дали 52 не е много? Можем да измислим и по-малко число, което ще е достатъчно, например половината на 52. Това също изглежда достатъчно, но ще е по-трудно да се обосновем защо.

Някои биха тръгнали с дебелите формули от теорията на вероятностите, но има ли смисъл? Числото 52 не е ли достатъчно малко, за да търсим по-малко. Цикъл, извършващ разцепването 52 пъти, минава мигновено, нали? Картите няма да са един милиард, нали? Следователно няма нужда да мислим в тази посока. Приемаме, че правим толкова "единични размествания", колкото са картите и това хем е достатъчно, хем не е прекалено много. Край, тази подзадача е решена.

Още един пример: сортиране на числа

Нека разгледаме накратко и още един пример. Даден е масив с числа и трябва да го сортираме по големина, т.е. да подредим елементите му в нарастващ ред. Това е задача, която има десетки концептуално различни методи за решаване и вие можете да измислите стотици идеи, някои от които са верни, а други – не съвсем.

Ако имаме тази задача и приемем, че е забранено да се ползват вградените в .NET Framework методи за сортиране, е нормално да вземем лист и химикал, да си направим един пример и да започнем да разсъждаваме. Можем да достигнем до много различни идеи, например:

- Първа идея: можем да изберем най-малкото число, да го отпечатаме и да го изтрием от масива. След това можем да повторим същото действие многократно докато масивът свърши. Разсъждавайки по тази идея, можем да разделим задачата на няколко по-прости задачки: намиране на най-малко число в масив; изтриване на число от масив; отпечатване на число.

- Следваща идея: можем да вземем най-малкото число и да го преместим най-отпред (чрез изтриване и вмъкване). След това в останалата част от масива можем пак да намерим най-малкото число и да го преместим веднага след първото. На k-тата стъпка ще имаме първите k най-малки числа в началото на масива. При този подход задачата се разделя по естествен начин на няколко по-малки задачки: намиране на най-малко число в част от масив и преместване на число от една позиция на масив в друга. Последната задачка може да се разбие на две по-малки: "изтриване на елемент от дадена позиция в масив" и "вмъкване на елемент в масив на дадена позиция").

- Поредна нова идея, която се базира на коренно различен подход: да разделим масива на две части с приблизително равен брой елементи, след което да сортираме първата част, да сортираме втората част и накрая да обединим двете части. Можем да приложим същото рекурсивно за всяка от частите докато не достигнем до част с големина един елемент, който очевидно е сортиран. При този подход пак имаме разделяне на сложната задача на няколко по-прости подзадачи: разделяне на масив на две равни (или почти равни) части; сливане на сортирани масиви.

Няма нужда да продължаваме повече, нали?. Всеки може да измисли още много идеи за решаване на задачата или да ги прочете в някоя книга по алгоритми. Показахме ви, че винаги сложната задача може да се раздели на няколко по-малки и по-прости задачки. Това е правилният подход при решаване на задачи по програмиране – да мислим за големия проблем като за съвкупност от няколко по-малки и по-прости проблема. Това е техника, която се усвоява бавно с времето, но рано или късно ще трябва да свикнете с нея.

Проверете идеите си!

Изглежда не остана нищо повече за измисляне. Имаме идея. Тя изглежда, че работи. Остава да проверим дали наистина работи или само така си мислим и след това да се ориентираме към имплементация.

Как да проверим идеята си? Обикновено това става с един или с няколко примера. Трябва да подберете такива примери, които в пълнота покриват различните случаи, които вашият алгоритъм трябва да преодолее. Примерите трябва хем да не са лесни за вашия алгоритъм, хем да са достатъчно прости, за да ги разпишете бързо и лесно. Такива примери наричаме "добри представители на общия случай".

Например ако реализираме алгоритъм за сортиране на масив в нарастващ ред, удачно е да вземем пример с 5-6 числа, сред които има 2 еднакви, а останалите са различни. Числата трябва първоначално да са подредени в случаен ред. Това е добър пример, понеже покрива много голяма част от случаите, в които нашият алгоритъм трябва да работи.

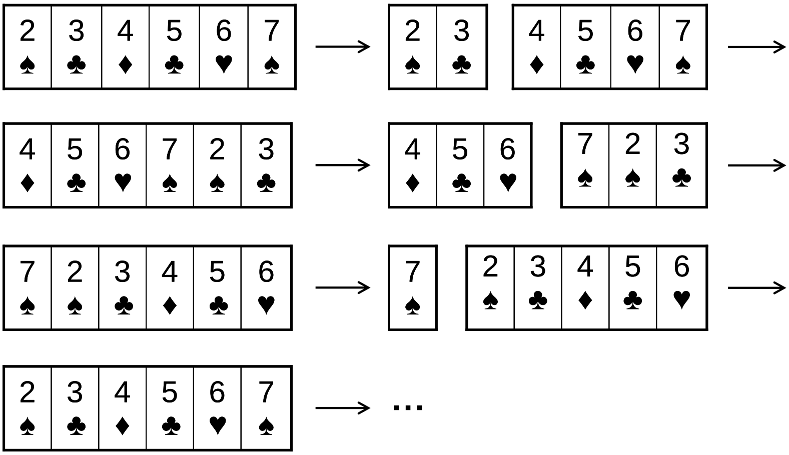
За същата задача за сортиране има множество неподходящи примери, с които няма да можете ефективно да проверите дали вашата идея за решение работи коректно. Например можем да вземем пример само с 2 числа. За него алгоритъмът може да работи, но по идея да е грешен. Можем да вземем пример само с еднакви числа. При него всеки алгоритъм за сортиране ще работи. Можем да вземем пример с числа, които са предварително подредени по големина. И за него алгоритъмът може да работи, но да е грешен.

Когато проверявате идеите си подбирайте подходящи примери. Те трябва хем да са прости и лесни за разписване, хем да не са частен случай, при който вашата идея би могла да работи, но да е грешна в общия случай. Примерите, които избирате, трябва да са добри представители на общия случай – да покриват възможно повече случаи, без да са големи и сложни.

Разбъркване на карти: проверка на идеята

Нека измислим един пример за нашата задача за разбъркване на карти, да кажем с 6 карти. За да е добър примерът, картите не трябва да са малко (да кажем 2-3), защото така примерът е прекалено лесен, но не трябва и да са много, за да можем бързо да проиграем нашата идея върху него. Добре е картите да са подредени първоначално по големина или даже за по-лесно да са поредни, за да може накрая лесно да видим дали са разбъркани – ако се запазят поредни или частично подредени, значи разбъркването не работи добре. Може би е най-хитро да вземем 6 карти, които са поредни, без значение на боята.

Вече измислихме пример, който е добър представител на общия случай за нашата задача. Нека да го нарисуваме на лист хартия и да проиграем върху него измисления алгоритъм. Трябва 6 пъти подред да сцепим на случайно място поредицата карти и да разменим получените 2 части. Нека картите първоначално са наредени по големина. Очакваме накрая картите да са случайно разбъркани. Да видим какво ще получим:



Няма нужда да правим 6 разцепвания. Вижда се, че след 3 размествания се върнахме в изходна позиция. Това едва ли е случайно. Какво стана? Открихме проблем в алгоритъма. Изглежда, че нашата идея е грешна. Като се замислим малко, се вижда, че всяко единично разместване през случайната позиция k всъщност ротира наляво тестето карти k пъти и след общо N ротации стигаме до изходна позиция. Добре, че тествахме на ръка алгоритъма преди да сме написали програмата, нали?

Сортиране на числа: проверка на идеята

Ако вземем проблема за сортирането на числа по големина и първия алгоритъм, който ни хрумна, можем лесно да проверим дали е верен. При него започваме с масив от N елемента и N пъти намираме в него най-малкото число, отпечатваме го и го изтриваме. Дори и без да я разписваме на хартия тази идея изглежда безпогрешна. Все пак нека вземем един пример и да видим какво ще се получи. Избираме 5 числа, като 2 от тях са еднакви: 3, 2, 6, 1, 2. Имаме 5 стъпки:

1) 3, 2, 6, 1, 2 → 1
2) 3, 2, 6, 2 → 2
3) 3, 6, 2 → 2
4) 3, 6 → 3
5) 6 → 6

Изглежда алгоритъмът работи коректно. Резултатът е верен и нямаме основание да си мислим, че няма да работи и за всеки друг пример.

При проблем измислете нова идея!

Нормално е, след като намерим проблем в нашата идея, да измислим нова идея, която би трябвало да работи. Това може да стане по два начина: или да поправим старата си идея, като отстраним дефектите в нея, или да измислим напълно нова идея. Нека видим как това работи за нашата задача за разбъркване на карти.

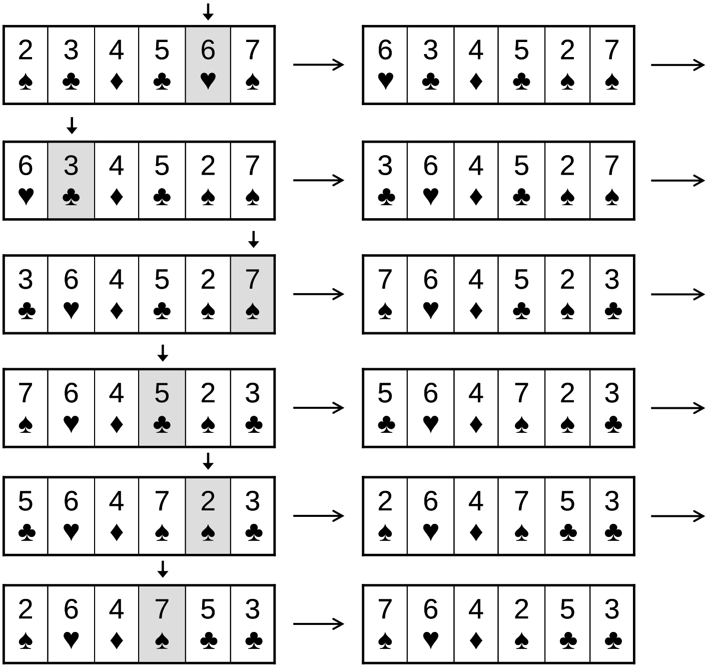
Измислянето на решение на задача по програмиране е итеративен процес, който включва последователно измисляне на идеи, изпробването им и евентуално замяната им с по-добри идеи при откриване на проблем. Понякога още първата идея е правилна, а понякога пробваме и отхвърляме една по една много различни идеи докато стигнем до такава, която да ни свърши работа.

Да се върнем на нашата задача за разбъркване на тесте карти. Първото нещо, което ни хрумва, е да видим защо е грешна нашата първа идея и да се опитаме да я поправим, ако това е възможно. Проблемът лесно се забелязва: последователното разцепване на тестето на две части и размяната им не води до случайна наредба на картите, а до някаква тяхна ротация (изместване наляво с някакъв брой позиции).

Как да поправим алгоритъма? Необходим ни е по-умен начин да правим единичното разместване, нали? Хрумва ни следната идея: взимаме две случайни карти и ги разменяме една с друга? Ако го направим N на брой пъти, сигурно ще се получи случайна наредба. Идеята изглежда по-добра от предната и може би работи. Вече знаем, че преди да мислим за реализация на новия алгоритъм трябва да го проверим дали работи правилно. Започваме да скицираме на хартия какво ще се случи за нашия пример с 6 карти.

В този момент ни хрумва нова, като че ли по-добра идея. Не е ли по-лесно на всяка стъпка да вземем случайна карта и да я разместим с първата? Изглежда по-просто и по-лесно за реализация, а резултатът би трябвало пак да е случаен. Първоначално ще разменим карта от случайна позиция k_1 с първата карта. Ще имаме случайна карта на първа позиция и първата карта ще бъде на позиция k_1. На следващата стъпка ще изберем случайна карта на позиция k_2 и ще я разменим с първата карта (картата от позиция k_1). Така вече първата карта си е сменила позицията, картата от позиция k_1 си е сменила позицията и картата от позиция k_2 също си е сменила позицията. Изглежда, че на всяка стъпка по една карта си сменя позицията със случайна. След такива N стъпки можем да очакваме всяка карта средно по веднъж да си е сменила мястото и следователно картите би трябвало да са добре разбъркани.

Дали това наистина е така? Да не стане като предния път? Нека проверим старателно тази идея. Отново можем да вземем 6 карти, които представляват добре подбран пример за нашата задача (добър представител на общия случай), и да ги разбъркаме по новия алгоритъм. Трябва да направим 6 последователни размествания на случайна карта с първата карта от тестето. Ето какво се получава:



От примера виждаме, че резултатът е правилен – получава се наистина случайно разбъркване на нашето примерно тесте от 6 карти. Щом нашият алгоритъм работи за 6 карти, би трябвало да работи и за друг брой. Ако не сме убедени в това, е хубаво да вземем друг пример, който изглежда по-труден за нашия алгоритъм.

Ако сме твърдо убедени, че идеята е вярна, може и да си спестим разписването на повече примери на хартия и направо продължим напред с решаването на задачата.

Да обобщим какво направихме до момента и как чрез последователни разсъждения стигнахме до идея за решаването на задачата. Следвайки всички препоръки, изложени до момента, минахме през следните стъпки:

- Използвахме лист и химикал, за да си скицираме тесте карти за разбъркване. Нарисувахме си последователност от кутийки на лист хартия и така успяхме визуално да си представим картите.
- Имайки визуална представа за проблема, ни хрумнаха някои идеи: първо, че трябва да правим някакви единични размествания и второ, че трябва да ги правим много на брой пъти.
- Решихме да правим единични размествания чрез цепене на картите на случайно място и размяна на двете половини.
- Решихме, че трябва да правим толкова размествания, колкото са картите в тестето.
- Сблъскахме се и с проблема за избор на случайно число, но избрахме решение наготово.
- Разбихме оригиналната задача на три подзадачи: единично разместване; избор на случайно число; повтаряне на единичните размествания.
- Проверихме дали идеята работи и намерихме грешка. Добре, че направихме проверка преди да напишем кода!
- Измислихме нова стратегия за единично разместване, която изглежда по-надеждна.
- Проверихме новата идея с подходящи примери и имаме увереност, че е правилна.

Вече имаме идея за решение на задачата и тя е проверена с примери. Това е най-важното за решаването на една задача – да измислим алгоритъма. Остава по-лесното – да реализираме идеята си. Нека видим как става това.

Подберете структурите от данни!

Ако вече имаме идея за решение, която изглежда правилна и е проверена с няколко надеждни примера, остава да напишем програмния код, нали?

Какво изпуснахме? Измислихме ли всичко необходимо, за да можем бързо, лесно и безпроблемно да напишем програма, която реализира нашата идея за решаване на задачата?

Това, което изпуснахме, е да си представим как нашата идея (която видяхме как работи на хартия) ще бъде имплементирана като компютърна програма. Това не винаги е елементарно и понякога изисква доста време и допълнителни идеи. Това е важна стъпка от решаването на задачи: да помислим за идеите си в термините на компютърното програмиране. Това означава да разсъждаваме с конкретни структури от данни, а не с абстракции като "карта" и "тесте карти". Трябва да подберем подходящи структури от данни, с които да реализираме идеите си.

Преди да преминете към имплементация на вашата идея помислете за структурите от данни. Може да се окаже, че вашата идея не е толкова добра, колкото изглежда. Може да се окаже, че е трудна за реализация или неефективна. По-добре да откриете това преди да сте написали кода на програмата!

В нашия случай говорихме за "размяна на случайна карта с друга", а в програмирането това означава да разместим два елемента в някаква структура от данни (например масив, списък или нещо друго). Стигнахме до момента, в който трябва да изберем структурите от данни и ще ви покажем как се прави това.

В каква структура да пазим тестето карти?

Първият въпрос, който възниква, е в каква структура от данни да съхраняваме тестето карти. Могат да ни хрумнат всякакви идеи, но не всички структури от данни са подходящи. Нека разсъждаваме малко по въпроса. Имаме съвкупност от карти и наредбата на картите в тази структура е от значение. Следователно трябва да използваме структура, която съхранява съвкупност от елементи и запазва наредбата им.

Можем ли да ползваме масив?

Първото, което можем да се сетим, е да използваме "масив". Това е най-простата структура за съхранение на съвкупност от елементи. Масивът може да съхранява съвкупност от елементи, в него елементите имат наредба (първи, втори трети и т.н.) и са достъпни по индекс. Масивът не може да променя първоначално определения му размер.

Подходяща структура ли е масивът? За да си отговорим на този въпрос, трябва да помислим какво трябва да правим с тестето карти, записано в масив и да проверим дали всяка от необходимите ни операции може да се реализира ефективно с масив.

Кои са операциите с тестето карти, които ще ни се наложи да реализираме за нашия алгоритъм? Нека ги изброим:

- Избор на случайна карта. Понеже в масива имаме достъп до елементите по индекс, можем да изберем случайно място в него чрез избор на случайно число k в интервала от 1 до N-1.
- Размяна на карта на позиция k с първата карта (единично разместване). След като сме избрали случайна карта, трябва да я разменим с първата. И тази операция изглежда проста. Можем да направим размяната на три стъпки чрез временна променлива.
- Въвеждане на тестето, обхождане на картите от тестето, отпечатване на тестето – всички тези операции биха могли да ни потрябват, но изглежда тривиално да ги реализираме с масив.

Изглежда, че обикновен масив може да ни свърши работа за съхранение на тесте карти.

Можем ли да ползваме друга структура?

Нормално е да си зададем въпроса дали масив е най-подходящата структура от данни за реализиране на операциите, които нашата програма трябва да извършва върху тестето карти. Изглежда, че всички операции могат лесно да се реализират с масив.

Все пак, нека помислим дали можем да изберем по-подходяща структура от масив. Нека помислим какви са възможностите ни:

- Свързан списък – нямаме директен достъп по номер на елемент и ще ни е трудно да избираме случайна карта от списъка.
- Масив с променлива дължина (**`List<T>`**) – изглежда, че притежава всички предимства на масивите и може да реализира всички операции, които ни трябват, по същия начин, както с масив. Печелим малко удобство – в **`List<T>`** можем лесно да трием и добавяме, което може да улесни въвеждането на картите и някои други помощни операции.
- Стек / опашка – тестето карти няма поведение на FIFO / LIFO и следователно тези структури не са подходящи.
- Множество (**`TreeSet<T>`** / **`HashSet<T>`**) – в множествата се губи оригиналната наредба на елементите и това е съществена пречка, за да ги използваме.
- Хеш-таблица – структурата "тесте карти" не е от вида ключ-стойност и следователно хеш-таблицата не може да го съхранява и обработва ефективно. Освен това хеш-таблиците не запазват подредбата на елементите си.

Общо взето изчерпахме основните структури от данни, които съхраняват и обработват съвкупности от елементи и стигнахме до извода, че масив или

`List<T>` ще ни свършат работа, а `List<T>` е по-гъвкав и удобен от обикновения масив. Взимаме решение да ползваме `List<T>` за съхранението и обработката на тестето карти.

> **Изборът на структура данни започва с изброяване на ключовите операции, които ще се извършват върху нея. След това се анализират възможните структури, които могат да бъдат използвани и от тях се избира тази, която най-лесно и ефективно реализира тези операции. Понякога се прави компромис между лесното на реализация и ефективност.**

Как да пазим другите информационни обекти?

След като решихме първия проблем, а именно как да представяме в паметта тесте от карти, следва да помислим дали има и други обекти, с които боравим, за които следва да помислим как да ги представяме. Като се замислим, освен обектите "карта" и "тесте карти", нашият алгоритъм не използва други информационни обекти.

Възниква въпросът как да представим една карта? Можем да я представим като символен низ, като число или като клас с две полета – лице и боя. Има, разбира се и други варианти, които имат своите предимства и недостатъци.

Преди да навлезем в дълбоки разсъждения кое представяне е най-добро, нека се върнем на условието на задачата. То предполага, че тестето карти ни е дадено (като масив или списък) и трябва да го разместим. Какво точно представлява една карта няма никакво значение за тази задача. Дори няма значение дали разместваме карти за игра, фигури за шах, кашони с домати или някакви други обекти. Имаме наредена последователност от обекти и трябва да я разбъркаме в случаен ред. Фактът, че разбъркваме карти, няма значение за нашата задача и няма нужда да губим време да мислим как точно да представим една карта. Нека просто се спрем на първата идея, която ни хрумва, примерно да си дефинираме клас `Card` с полета `Face` и `Suit`. Дори да изберем друго представяне (примерно число от 1 до 52), това не е съществено. Няма да дискутираме повече този въпрос.

Сортиране на числа – подбор на структурите данни

Нека се върнем на задачата за сортиране на съвкупност от числа по големина и изберем структури от данни и за нея. Нека сме избрали да използваме най-простия алгоритъм, за който сме се сетили: да взимаме докато може най-малкото число, да го отпечатваме и да го изтриваме. Тази идея лесно се разписва на хартия и лесно се убеждаваме, че е коректна.

Каква структура от данни да използваме за съхранение на числата? Отново, за да си отговорим на този въпрос, е необходимо да помислим какви операции трябва да извършваме върху тези числа. Операциите са следните:

- Търсене на най-малка стойност в структурата.
- Изтриване на намерената най-малка стойност от структурата.

Очевидно използването на масив не е разумно, защото не разполагаме с операцията "изтриване". Използването на **`List<T>`** изглежда по-добре, защото и двете операции можем да реализираме сравнително просто и лесно. Структури като стек и опашка няма да ни помогнат, защото нямаме LIFO или FIFO поведение. От хеш-таблица няма особен смисъл, защото в нея няма бърз начин за намиране на най-малка стойност, въпреки че изтриването на елемент би могло да е по-ефективно.

Стигаме до структурите **`HashSet<T>`** и **`TreeSet<T>`**. Множествата имат проблема, че не поддържат възможност за съхранение на еднакви елементи. Въпреки това, нека ги разгледаме. Структурата **`HashSet<T>`** не представлява интерес, защото при нея отново нямаме лесен начин да намерим най-малкия елемент. Обаче структурата **`TreeSet<T>`** изглежда обещаваща. Нека я разгледаме.

Класът **`TreeSet<T>`** по идея държи елементите си в балансирано дърво и поддържа операцията "изваждане на най-малкия елемент". Колко интересно! Хрумва ни нова идея: вкарваме всички елементи в **`TreeSet<T>`** и изкарваме от него итеративно най-малкия елемент докато елементите свършат. Просто, лесно и ефективно. Имаме наготово двете операции, които ни интересуват (търсене на най-малък елемент и изтриването му от структурата).

Докато си представяме конкретната имплементация и се ровим в документацията се сещаме нещо още по-интересно: класът **`TreeSet<T>`** държи вътрешно елементите си подредени по големина. Ами нали това се иска в задачата: да наредим елементите по големина. Следователно, ако ги вкараме в **`TreeSet<T>`** и след това обходим елементите му (чрез неговия итератор), те ще бъдат подредени по големина. Задачата е решена!

Докато се радваме, се сещаме за един забравен проблем: **`TreeSet<T>`** не поддържа еднакви елементи, т.е. ако имаме числото 5 няколко пъти, то ще се появи в множеството само веднъж. В крайна сметка при сортирането ще загубим безвъзвратно някои от елементите.

Естествено е да потърсим решение на този проблем. Ако има начин да пазим колко пъти се среща всеки елемент от множеството, това ще ни реши проблема. Тогава се сещаме за класа **`SortedDictionary<K,T>`**. Той съхранява множество ключове, които са подредени по големина и във всеки ключ можем да имаме стойност. В стойността можем да съхраняваме колко пъти се среща даден елемент. Можем да преминем с един цикъл през елементите на масива и за всеки от тях да запишем колко пъти се среща в структура `SortedDictionary<K,T>`. Изглежда това решава проблема ни и можем да го реализираме, макар и не толкова лесно, колкото с `List<T>` или с `TreeSet<T>`.

Ако прочетем внимателно документацията за `SortedDictionary<K,T>`, ще се убедим, че този клас вътрешно използва червено-черно дърво и може някой ден да се досетим, че неусетно чрез разсъждения сме достигнали до добре известния алгоритъм "сортиране чрез дърво" (http://en.wikipedia.org/wiki/Binary_tree_sort).

Видяхте до какви идеи ви довеждат разсъжденията за избор на подходящи структури от данни за имплементация на вашите идеи. Тръгвате от един алгоритъм и неусетно измисляте нов, по-добър. Това е нормално да се случи в процеса на обмисляне на алгоритъма и е добре да се случи в този момент, а не едва когато сте написали вече 300 реда код, който ще се наложи да преправяте. Това е още едно доказателство, че трябва да помислите за структурите от данни преди да започнете да пишете кода.

Помислете за ефективността!

За пореден път изглежда, че най-сетне сме готови да хванем клавиатурата и да напишем кода на програмата. И за пореден път е добре да не избързваме. Причината е, че не сме помислили за нещо много важно: ефективност и бързодействие.

⚠	**За ефективността трябва да се помисли още преди да се напише първия ред програмен код! Иначе рискувате да загубите много време за реализация на идея, която не върши работа!**

Да се върнем на задачата за разбъркване на тесте карти. Имаме идея за решаване на задачата (измислили сме алгоритъм). Идеята изглежда коректна (пробвали сме я с примери). Идеята изглежда, че може да се реализира (ще ползваме `List<Card>` за тестето карти и клас `Card` за представянето на една карта). Обаче, нека помислим колко карти ще разбъркваме и дали избраната идея, реализирана с избраните структури от данни, ще работи достатъчно бързо.

Как оценяваме бързината на даден алгоритъм?

Бърз ли е нашият алгоритъм? За да си отговорим на този въпрос, нека помислим колко операции извършва той за разбъркването на стандартно тесте от 52 карти.

За 52 карти нашият алгоритъм прави 52 единични размествания, нали така? Колко елементарни операции отнема едно единично разместване? Операциите са 4: избор на случайна карта; запазване на първата карта във временна променлива; запис на случайната карта на мястото на първата; запис на първата карта (от временната променлива) на мястото, където е била случайната карта. Колко операции прави общо нашият алгоритъм за 52 карти? Операциите са приблизително 52 * 4 = 208.

Много операции ли са 208? Замислете се колко време отнема да завъртите цикъл от 1 до 208. Много ли е? Пробвайте! Ще се убедите, че цикъл от 1 до 1 000 000 при съвременните компютри минава неусетно бързо, а цикъл до 208 отнема смешно малко време. Следователно нямаме проблем с производителността. Нашия алгоритъм ще работи супер бързо за 52 карти.

Въпреки, че в реалността рядко играем с повече от 1 или 2 тестета карти, нека се замислим колко време ще отнеме да разбъркаме голям брой карти, да кажем 50 000? Ще имаме 50 000 единични размествания по 4 операции за всяко от тях или общо 200 000 операции, които ще се изпълнят на момента, без да се усети каквото и да е забавяне.

Ефективността е въпрос на компромис

В крайна сметка правим извода, че алгоритъмът, който сме измислили е ефективен и ще работи добре дори при голям брой карти. Имахме късмет. Обикновено нещата не са толкова прости и трябва да се прави компромис между бързодействие на алгоритъма и усилията, които влагаме, за да го измислим и имплементираме. Например ако сортираме числа, можем да го направим за 5 минути с първия алгоритъм, за който се сетим, но можем да го направим и много по-ефективно, за което ще употребим много повече време (да търсим и да четем из дебелите книги и в Интернет). В този момент трябва да се прецени струва ли си усилията. Ако ще сортираме 20 числа, няма значение как ще го направим, все ще е бързо, дори с най-глупавия алгоритъм. Ако сортираме 20 000 числа вече алгоритъмът има значение, а ако сортираме 20 000 000 числа, задачата придобива съвсем друг характер. Времето, необходимо да реализираме ефективно сортиране на 20 000 000 числа е далеч повече от времето да сортираме 20 числа, така че трябва да помислим струва ли си.

Ефективността е въпрос на компромис – понякога не си струва да усложняваме алгоритъма и да влагаме време и усилия, за да го направим по-бърз, а друг път бързината е ключово изискване и трябва да й обърнем сериозно внимание.

Сортиране на числа – оценяване на ефективността

Видяхме, че подходът към въпроса с ефективността силно зависи от изискванията за бързодействие. Нека се върнем сега на задачата за сортирането на числа, защото искаме да покажем, че ефективността е пряко свързана с избора на структури от данни.

Да се върнем отново на въпроса за избор на структура от данни за съхранение на числата, които трябва да сортираме по големина в нарастващ ред. Дали да изберем **`List<T>`** или **`SortedDictionary<K,T>`**? Не е ли по-добре да ползваме някаква проста структура, която добре познаваме, отколкото някоя сложна, която изглежда, че ще ни свърши работата малко по-добре. Вие познавате ли добре червено-черните дървета (вътрешната имплементация на **`SortedDictionary<K,T>`**)? С какво са по-добри от **`List<T>`**? Всъщност може да се окаже, че няма нужда да си отговаряте на този въпрос.

Ако трябва да сортирате 20 числа, има ли значение как ще го направите? Взимате първия алгоритъм, за който се сетите, взимате първата структура от данни, която изглежда, че ще ви свърши работа и готово. Няма никакво значение колко са бързи избраните алгоритми и структури от данни, защото числата са изключително малко.

Ако, обаче трябва да сортирате 300 000 числа, нещата са съвсем различни. Тогава ще трябва внимателно да проучите как работи класът **`SortedDictionary<K,T>`** и колко бързо става добавянето и търсенето в него, след което ще трябва да оцените ориентировъчно колко операции ще са нужни за 300 000 добавяния на число и след това колко още операции ще отнеме обхождането. Ще трябва да прочетете документацията, където пише, че добавянето отнема средно $\log_2(N)$ стъпки, където N е броят елементи в структурата. Чрез дълги и мъчителни сметки (за които ви трябват допълнителни умения) може да оцените грубо, че ще са необходими около 5-6 милиона стъпки за цялото сортиране, което е приемливо бързо.

По аналогичен път, можете да се убедите, че търсенето и изтриването в **`List<T>`** с N елемента отнема N стъпки и следователно за 300 000 елемента ще ни трябват приблизително 2 * 300 000 * 300 000 стъпки! Всъщност това число е силно закръглено нагоре, защото в началото нямате 300 000 числа, а само 1, но грубата оценка е пак приблизително вярна. Получава се екстремално голям брой стъпки и простичкият алгоритъм няма да работи за такъв голям брой елементи (програмата мъчително ще "увисне").

Отново стигаме до въпроса с компромиса между сложния и простия алгоритъм. Единият е по-лесен за имплементиране, но е по-бавен. Другият е по-ефективен, но е по-сложен за имплементиране и изисква да четем документация и дебели книги, за да разберем колко бързо ще работи. Въпрос на компромис.

Естествено, в този момент можем да се сетим за някоя от другите идеи за сортиране на числа, които ни бяха хрумнали в началото, например идеята да разделим масива на две части, да ги сортираме поотделно (чрез рекурсивно извикване) и да ги слеем в един общ масив. Ако помислим, ще се убедим, че този алгоритъм може да се реализира ефективно с обикновен динамичен масив (**`List<T>`**) и че той прави в най-лошия случай n*log(n) стъпки при n елемента, т.е. ще работи добре за 300 000 числа. Няма да навлизаме повече в детайли, тъй като всеки може да прочете за MergeSort в Уикипедия (http://en.wikipedia.org/wiki/Merge_sort).

Имплементирайте алгоритъма си!

Най-сетне стигаме до имплементация на нашата идея за решаване на задачата. Вече имаме работеща и проверена идея, подбрали сме подходящи структури от данни и остава да напишем кода. Ако не сме направили това, трябва да се върнем на предните стъпки.

Ако нямате измислена идея за решение, не започвайте да пишете код! Какво ще напишете, като нямате идея за решаване на задачата? Все едно да отидете на гарата и да се качите на някой влак, без да сте решили за къде ще пътувате.

Типично действие за начинаещите програмисти: като видят задачата да почнат веднага да пишат и след като загубят няколко часа в писане на необмислени идеи (които им хрумват докато пишат), да се сетят да помислят малко. Това е грешно и целта на всички препоръки до момента е да ви предпази от такъв лекомислен и крайно неефективен подход.

Ако не сте проверили дали идеите ви са верни, не почвайте да пишете код! Трябва ли да напишете 300 реда код и тогава да откриете, че идеята ви е тотално сбъркана и трябва да почнете отначало?

Писането на кода при вече измислена и проверена идея изглежда просто и лесно, но и за него се изискват специфични умения и най-вече опит. Колкото повече програмен код сте писали, толкова по-бързо, ефективно и без грешки се научавате да пишете. С много практика ще постигнете лекота при писането и постепенно с времето ще се научите да пишете не само бързо, но и качествено. За качеството на кода можете да прочетете в главата "Качествен програмен код", така че, нека се фокусираме върху правилния процес за написването на кода.

Считаме, че би трябвало вече да сте овладели начални техники, свързани с писането на програмен код: как да работите със средата за разработка (Visual Studio), как да ползвате компилатора, как да разчитате грешките, които той ви дава, как да ползвате подсказките (auto complete), как да генерирате методи, конструктори и свойства, как да поправяте грешки и как да изпълнявате и дебъгвате програмата. Затова съветите, които следват, са свързани не със самото писане на програмни редове код, а с цялостния подход при имплементиране на алгоритми.

Пишете стъпка по стъпка!

Случвало ли ви се е да напишете 200-300 реда код, без да опитате поне веднъж да компилирате и да тествате дали нещо работи? Не правете така! Не пишете много код на един път, а вместо това пишете стъпка по стъпка.

Как да пишем стъпка по стъпка? Това зависи от конкретната задача и от начина, по който сме я разделили на подзадачи. Например ако задачата се състои от 3 независими части, напишете първо едната част, компилирайте я, тествайте я с някакви примерни входни данни и след като се убедите, че работи, преминете към следващите части. След това напишете втората част, компилирайте я, тествайте я и когато и тя е готова, преминете към третата част. Когато сте написали и последната част и сте се убедили, че работи правилно, преминете към обстойно тестване на цялата програма.

Защо да пишем на части? Когато пишете на части, стъпка по стъпка, вие намалявате обема код, над който се концентрирате във всеки един момент. По този начин намалявате сложността на проблема, като го разглеждате на части. Спомнете си: големият и сложен проблем винаги може да се раздели на няколко по-малки и по-прости проблема, за които лесно ще намерите решение.

Когато напишем голямо количество код, без да сме опитали да компилираме поне веднъж, се натрупват голямо количество грешки, които могат да се избегнат чрез просто компилиране. Съвременните среди за програмиране (като Visual Studio) се опитват да откриват синтактичните грешки автоматично още докато пишете кода. Ползвайте тази възможност и отстранявайте грешките възможно най-рано. Ранното отстраняване на проблеми отнема по-малко време и нерви. Късното отстраняване на грешки и проблеми може да коства много усилия, дори понякога и цялостно пренаписване на програмата.

Когато напишете голямо количество код, без да го тествате и след това решите наведнъж да го изпробвате за някакви примерни входни данни, обикновено се натъквате на множество проблеми, изсипващи се един след друг, като колкото повече е кодът, толкова по-трудно е те да бъдат оправени. Проблемите могат да са причинени от необмислено използване на неподходящи структури от данни, грешен алгоритъм, необмислено структуриране на кода, грешно условие в `if`-конструкция, грешно организиран цикъл, излизане извън граници на масив и много, много други проблеми, които е можело да бъдат отстранени много по-рано и с много по-малко усилия. Затова не чакайте последния момент. Отстранявайте грешките възможно най-рано!

Пишете програмата на части, а не наведнъж! Напишете някаква логически отделена част, компилирайте я, отстранете грешките, тествайте я и когато тя работи, преминете

	към следващата част.

Писане стъпка по стъпка – пример

За да илюстрираме на практика как можем да пишем стъпка по стъпка, нека се захванем с имплементация на алгоритъма за разбъркване на карти, който измислихме следвайки препоръките за решаване на алгоритмични задачи, описани по-горе.

Стъпка 1 – Дефиниране на клас "карта"

Тъй като трябва да разбъркваме карти, можем да започнем с дефиницията на класа "карта". Ако нямаме идея как да представяме една карта, няма да имаме и как да представяме тесте карти, следователно няма да има и как да дефинираме метода за разбъркване на картите. Вече споменахме, че представянето на картите не е от значение за поставената задача, така че всякакво представяне би ни свършило работа.

Ще дефинираме клас "карта" с полета лице и боя. Ще използваме символен низ за лицето (с възможни стойности "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K" или "A") и изброен тип за боята (с възможни стойности "спатия", "каро", "купа" и "пика"). Класът `Card` би могъл да изглежда по следния начин:

Card.cs

```
class Card
{
  public string Face { get; set; }
  public Suit Suit { get; set; }

  public override string ToString()
  {
    String card = "(" + this.Face + " " + this.Suit +")";
    return card;
  }
}

enum Suit
{
  CLUB, DIAMOND, HEART, SPADE
}
```

За удобство дефинирахме и метод **`ToString()`** в класа **`Card`**, с който можем по-лесно да отпечатваме дадена карта на конзолата. За боите дефинирахме изброен тип **`Suit`**.

Изпробване на класа "карта"

Някои от вас биха продължили да пишат напред, но следвайки принципа "програмиране стъпка по стъпка", трябва първо да тестваме дали класът **`Card`** се компилира и работи правилно. За целта можем да си направим малка програмка, в която създаваме една карта и я отпечатваме:

```
static void Main()
{
   Card card = new Card() { Face="A", Suit=Suit.CLUB };
   Console.WriteLine(card);
}
```

Стартираме програмката и проверяваме дали картата се е отпечатала коректно. Резултатът е следният:

```
(A CLUB)
```

Стъпка 2 – Създаване и отпечатване на тесте карти

Нека преди да преминем към същината на задачата (разбъркване на тесте карти в случаен ред) се опитаме да създадем цяло тесте от 52 карти и да го отпечатаме. Така ще се убедим, че входът на метода за разбъркване на карти е коректен. Според направения анализ на структурите данни, трябва да използваме **`List<Card>`**, за да представяме тестето. Нека създадем тесте от 5 карти и да го отпечатаме:

CardsShuffle.cs

```
class CardsShuffle
{
   static void Main()
   {
      List<Card> cards = new List<Card>();
      cards.Add(new Card() { Face = "7", Suit = Suit.HEART });
      cards.Add(new Card() { Face = "A", Suit = Suit.SPADE });
      cards.Add(new Card() { Face = "10", Suit = Suit.DIAMOND });
      cards.Add(new Card() { Face = "2", Suit = Suit.CLUB });
      cards.Add(new Card() { Face = "6", Suit = Suit.DIAMOND });
      cards.Add(new Card() { Face = "J", Suit = Suit.CLUB });
      PrintCards(cards);
   }

   static void PrintCards(List<Card> cards)
   {
      foreach (Card card in cards)
      {
         Console.Write(card);
      }
      Console.WriteLine();
   }
}
```

Отпечатване на тестето – тестване на кода

Преди да продължим напред, стартираме програмата и проверяваме дали сме получили очаквания резултат. Изглежда, че няма грешки и резултатът е коректен:

```
(7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6 DIAMOND)(J CLUB)
```

Стъпка 3 – Единично разместване

Нека реализираме поредната стъпка от решаването на задачата – подзадачата за единично разместване. Когато имаме логически отделена част от програмата е добра идея да я реализираме като отделен метод. Да помислим какво приема методът като вход и какво връща като изход. Като вход би трябвало да приема тесте карти (**`List<Card>`**). В резултат от работата си методът би трябвало да промени подадения като вход **`List<Card>`**. Методът няма нужда да връща нищо, защото не създава нов **`List<Card>`** за резултата, а оперира върху вече създадения и подаден като параметър списък.

Какво име да дадем на метода? Според препоръките за работа с методи трябва да дадем "описателно" име – такова, което описва с 1-2 думи какво прави метода. Подходящо за случая е името **`PerformSingleExchange`**. Името ясно описва какво прави методът: извършва единично разместване.

Нека първо дефинираме метода, а след това напишем тялото му. Това е добра практика, тъй като преди да започнем да реализираме даден метод трябва да сме наясно какво прави той, какви параметри приема, какъв резултат връща и как се казва. Ето как изглежда дефиницията на метода:

```
static void PerformSingleExchange(List<Card> cards)
{
   // TODO: Implement the method body
}
```

Следва да напишем тялото на метода. Първо трябва да си припомним алгоритъма, а той беше следният: избираме случайно число k в интервала от 1 до дължината на масива минус 1 и разменяме първия елемент на масива с k-тия елемент. Изглежда просто, но как в C# получаваме случайно число в даден интервал?

Търсете в Google!

Когато се натъкнем на често срещан проблем, за който нямаме решение, но знаем, че много хора са се сблъсквали с него, най-лесният начин да се справим е да потърсим информация в Google. Трябва да формулираме по подходящ начин нашето търсене. В случая търсим примерен C# код, който връща случайно число в даден интервал. Можем да пробваме следното търсене:

```
C# random number example
```

Сред първите резултати излиза C# програмка, която използва класа **`System.Random`**, за да генерира случайно число. Вече имаме посока, в която да търсим решение – знаем, че в .NET Framework има стандартен клас **`Random`**, който служи за генериране на случайни числа.

След това можем да се опитаме да налучкаме как се ползва този клас (често пъти това отнема по-малко време, отколкото да четем документацията). Опитваме да намерим подходящ статичен метод за случайно число, но се оказва, че такъв няма. Създаваме инстанция и търсим метод, който да ни върне число в даден диапазон. Имаме късмет, методът **`Next(minValue, maxValue)`** е връща каквото ни трябва.

Да опитаме да напишем кода на целия метод. Получава се нещо такова:

```
static void PerformSingleExchange(List<Card> cards)
{
   Random rand = new Random();
   int randomIndex = rand.Next(1, cards.Count - 1);
   Card firstCard = cards[1];
   Card randomCard = cards[randomIndex];
   cards[1] = randomCard;
   cards[randomIndex] = firstCard;
}
```

Единично разместване – тестване на кода

Следва тестване на кода. Преди да продължим нататък, трябва да се убедим, че единичното разместване работи коректно. Нали не искаме да открием евентуален проблем, едва когато тестваме метода за разбъркване на цялото тесте? Искаме, ако има проблем, да го открием веднага, а ако няма проблем, да се убедим в това, за да продължим уверено напред. Действаме стъпка по стъпка – преди да започнем следващата стъпка, проверяваме дали текущата е реализирана коректно. За целта си правим малка тестова програмка, да кажем с три карти (2♣, 3♥ и 4♠):

```
static void Main()
{
   List<Card> cards = new List<Card>();
   cards.Add(new Card() { Face = "2", Suit = Suit.CLUB });
   cards.Add(new Card() { Face = "3", Suit = Suit.HEART });
   cards.Add(new Card() { Face = "4", Suit = Suit.SPADE });
   PerformSingleExchange(cards);
   PrintCards(cards);
}
```

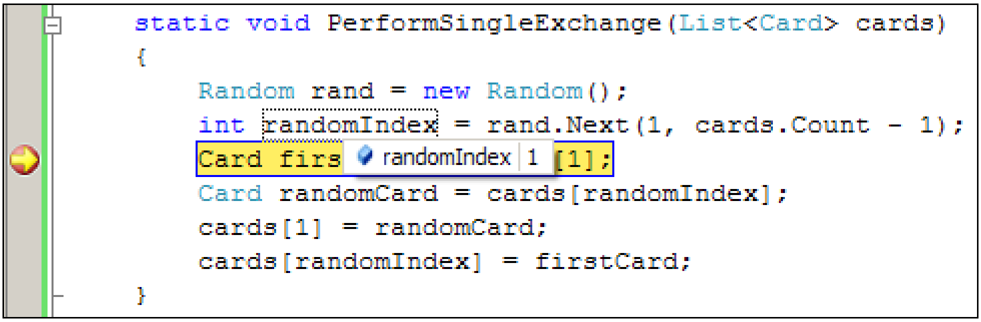
Нека изпълним няколко пъти единичното разместване с нашите 3 карти. Очакваме първата карта (двойката) да бъде разменена с някоя от другите две карти (с тройката или с четворката). Ако изпълним програмата много пъти, би следвало около половината от получените резултати да съдържат (3♥, 2♣, 4♠), а останалите – (4♠, 3♥, 2♣), нали така? Да видим какво ще получим. Стартираме програмата и получаваме следния резултат:

```
(2 CLUB)(3 HEART)(4 SPADE)
```

Ама как така? Какво стана? Да не сме забравили да изпълним единичното разместване преди да отпечатам картите? Има нещо гнило тук. Изглежда програмата не е направила нито едно разместване на нито една карта. Как стана тая работа?

Единично разместване – поправяне на грешките

Очевидно имаме грешка. Да сложим точка на прекъсване и да проследим какво се случва чрез дебъгера на Visual Studio:



```
static void PerformSingleExchange(List<Card> cards)
{
    Random rand = new Random();
    int randomIndex = rand.Next(1, cards.Count - 1);
    Card firs                [1];
    Card randomCard = cards[randomIndex];
    cards[1] = randomCard;
    cards[randomIndex] = firstCard;
}
```

Видно е, че при първо стартиране случайната позиция се случва да има стойност 1. Това е допустимо, така че продължаваме напред. Като погледнем кода малко по-надолу, виждаме, че разменяме случайния елемент с индекс 1 с елемента на позиция 1, т.е. със себе си. Очевидно нещо бъркаме. Сещаме се, че индексирането в **`List<T>`** започва от 0, а не от 1, т.е. първият елемент е на позиция 0. Веднага поправяме кода:

```
static void PerformSingleExchange(List<Card> cards)
{
   Random rand = new Random();
```

```
    int randomIndex = rand.Next(1, cards.Count - 1);
    Card firstCard = cards[0];
    Card randomCard = cards[randomIndex];
    cards[0] = randomCard;
    cards[randomIndex] = firstCard;
}
```

Стартираме програмата няколко пъти и получаваме пак странен резултат:

```
(3 HEART)(2 CLUB)(4 SPADE)
(3 HEART)(2 CLUB)(4 SPADE)
(3 HEART)(2 CLUB)(4 SPADE)
```

Изглежда случайното число не е съвсем случайно. Какво има пък сега? Не бързайте да обвинявате .NET Framework, CLR, Visual Studio и всички други заподозрени виновници! Може би грешката е отново при нас. Да разгледаме извикването на метода **`Next(…)`**. Понеже **`cards.Count`** е 3, то винаги викаме **`NextInt(1, 2)`** и очакваме да ни върне число между 1 и 2. Звучи коректно, обаче ако прочетем какво пише в документацията за метода **`Next(…)`**, ще забележим, че вторият параметър трябва да е с единица по-голям от максималното число, което искаме да получим.

Сбъркали сме с единица диапазона на случайната карта, която избираме. Поправяме кода и за пореден път тестваме дали работи. След втората поправка получаваме следната реализация на метода за единично разместване:

```
static void PerformSingleExchange(List<Card> cards)
{
    Random rand = new Random();
    int randomIndex = rand.Next(1, cards.Count);
    Card firstCard = cards[0];
    Card randomCard = cards[randomIndex];
    cards[0] = randomCard;
    cards[randomIndex] = firstCard;
}
```

Ето какво би могло да се получи след няколко изпълнения на горния метод върху нашата поредица от три карти:

```
(3 HEART)(2 CLUB)(4 SPADE)
(4 SPADE)(3 HEART)(2 CLUB)
(4 SPADE)(3 HEART)(2 CLUB)
(3 HEART)(2 CLUB)(4 SPADE)
(4 SPADE)(3 HEART)(2 CLUB)
(3 HEART)(2 CLUB)(4 SPADE)
```

Вижда се, че след достатъчно изпълнения на метода на мястото на първата карта отива всяка от следващите две карти, т.е. наистина имаме случайно разместване и всяка карта освен първата има еднакъв шанс да бъде избрана като случайна. Най-накрая сме готови с метода за единично разместване. Хубаво беше, че открихме двете грешки сега, а не по-късно, когато очакваме цялата програма да заработи.

Стъпка 4 – Разместване на тестето

Последната стъпка е проста: прилагаме N пъти единичното разместване:

```
static void ShuffleCards(List<Card> cards)
{
   for (int i = 1; i <= cards.Count; i++)
   {
      PerformSingleExchange(cards);
   }
}
```

Ето как изглежда цялата програма:

CardsShuffle.cs

```
using System;
using System.Collections.Generic;

class CardsShuffle
{
   static void Main()
   {
      List<Card> cards = new List<Card>();
      cards.Add(new Card() { Face = "2", Suit = Suit.CLUB });
      cards.Add(new Card() { Face = "6", Suit = Suit.DIAMOND });
      cards.Add(new Card() { Face = "7", Suit = Suit.HEART });
      cards.Add(new Card() { Face = "A", Suit = Suit.SPADE });
      cards.Add(new Card() { Face = "J", Suit = Suit.CLUB });
      cards.Add(new Card() { Face = "10", Suit = Suit.DIAMOND });

      Console.Write("Initial deck:  ");
      PrintCards(cards);

      ShuffleCards(cards);
      Console.Write("After shuffle: ");
      PrintCards(cards);
   }

   static void PerformSingleExchange(List<Card> cards)
   {
      Random rand = new Random();
```

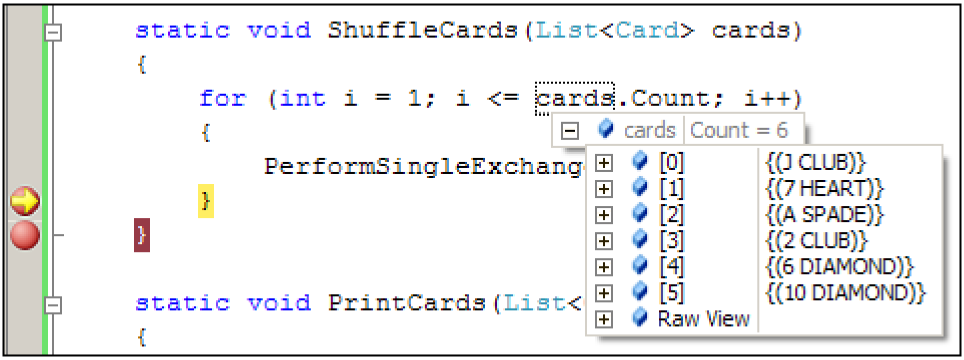
```
        int randomIndex = rand.Next(1, cards.Count);
        Card firstCard = cards[0];
        Card randomCard = cards[randomIndex];
        cards[0] = randomCard;
        cards[randomIndex] = firstCard;
    }

    static void ShuffleCards(List<Card> cards)
    {
        for (int i = 1; i <= cards.Count; i++)
        {
            PerformSingleExchange(cards);
        }
    }

    static void PrintCards(List<Card> cards)
    {
        foreach (Card card in cards)
        {
            Console.Write(card);
        }
        Console.WriteLine();
    }
}
```

Разместване на тестето – тестване

Остава да пробваме дали целият алгоритъм работи. Ето какво се получава след изпълнение на програмата:

```
Initial deck:  (7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6
DIAMOND)(J CLUB)
After shuffle: (7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6
DIAMOND)(J CLUB)
```

Очевидно отново имаме проблем: тестето карти след разбъркването е в началния си вид. Дали не сме забравили да извикаме метода за разбъркване **`ShuffleCards`**? Поглеждаме внимателно кода: всичко изглежда наред. Решаваме да сложим точка на прекъсване (breakpoint) веднага след извикването на метода **`PerformSingleExchange(…)`** в тялото на цикъла за разбъркване на картите. Стартираме програмата в режим на постъпково изпълнение (дебъгване) с натискане на **[F5]**. След първото спиране на дебъгера в точката на прекъсване всичко е наред – първата карта е разменена със случайна карта, точно както трябва да стане. След второто спиране на дебъгера отново всичко е наред – случайна карта е разменена с първата. Странно, изглежда, че всичко работи както трябва:



```
static void ShuffleCards(List<Card> cards)
{
    for (int i = 1; i <= cards.Count; i++)
    {
        PerformSingleExchang
    }
}

static void PrintCards(List<
{
```

```
cards | Count = 6
[0]       {(J CLUB)}
[1]       {(7 HEART)}
[2]       {(A SPADE)}
[3]       {(2 CLUB)}
[4]       {(6 DIAMOND)}
[5]       {(10 DIAMOND)}
Raw View
```

Защо тогава накрая резултатът е грешен? Решаваме да сложим точка на прекъсване и в края на метода **ShuffleCards(…)**. Дебъгерът спира и на него и отново резултатът в момента на прекъсване на програмата е какъвто трябва да бъде – картите са случайно разбъркани. Продължаваме да дебъгваме и стигаме до отпечатването на тестето карти. Преминаваме и през него и на конзолата се отпечатва разбърканото в случаен ред тесте карти. Странно: изглежда всичко работи. Какъв е проблемът?

Стартираме програмата без да я дебъгваме с **[Ctrl+F5]**. Резултатът е грешен – картите не са разбъркани. Стартираме програмата отново в режим на дебъгване с **[F5]**. Дебъгерът отново спира на точките на прекъсване и отново програмата се държи коректно. Изглежда, че когато дебъгваме програмата, тя работи коректно, а когато я стартираме без дебъгер, резултатът е грешен. Странна работа!

Решаваме да добавим един ред, който отпечатва тестето карти след всяко единично разместване:

```
static void ShuffleCards(List<Card> cards)
{
    for (int i = 1; i <= cards.Count; i++)
    {
        PerformSingleExchange(cards);
        PrintCards(cards);
    }
}
```

Стартираме програмата през дебъгера (с **[F5]**), проследяваме постъпково нейното изпълнение и установяваме, че работи правилно:

```
Initial deck:  (7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6
DIAMOND)(J CLUB)
(A SPADE)(7 HEART)(10 DIAMOND)(2 CLUB)(6 DIAMOND)(J CLUB)
(6 DIAMOND)(7 HEART)(10 DIAMOND)(2 CLUB)(A SPADE)(J CLUB)
(J CLUB)(7 HEART)(10 DIAMOND)(2 CLUB)(A SPADE)(6 DIAMOND)
(2 CLUB)(7 HEART)(10 DIAMOND)(J CLUB)(A SPADE)(6 DIAMOND)
(A SPADE)(7 HEART)(10 DIAMOND)(J CLUB)(2 CLUB)(6 DIAMOND)
```

```
(10 DIAMOND)(7 HEART)(A SPADE)(J CLUB)(2 CLUB)(6 DIAMOND)
After shuffle: (10 DIAMOND)(7 HEART)(A SPADE)(J CLUB)(2 CLUB)(6
DIAMOND)
```

Стартираме отново програмата без дебъгера (с **[Ctrl+F5]**) и получаваме отново грешния резултат, който се опитваме да разберем как и защо се получава:

```
Initial deck:  (7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6
DIAMOND)(J CLUB)
(6 DIAMOND)(A SPADE)(10 DIAMOND)(2 CLUB)(7 HEART)(J CLUB)
(7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6 DIAMOND)(J CLUB)
(6 DIAMOND)(A SPADE)(10 DIAMOND)(2 CLUB)(7 HEART)(J CLUB)
(7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6 DIAMOND)(J CLUB)
(6 DIAMOND)(A SPADE)(10 DIAMOND)(2 CLUB)(7 HEART)(J CLUB)
(7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6 DIAMOND)(J CLUB)
After shuffle: (7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6
DIAMOND)(J CLUB)
```

Непосредствено се вижда, че на всяка стъпка, в която се очаква да се извърши единично разместване, реално се разместват едни и същи карти: 7♥ и 6♦. Има само един начин това да се случи – ако всеки път случайното число, което се пада, е едно и също. Изводът е, че нещо не е наред с генерирането на случайни числа. Веднага ни хрумва да погледнем документацията на класа **`System.Random()`**. В MSDN можем да прочетем, че при създаване на нова инстанция на генератора на псевдослучайни числа с конструктора **`Random()`** генераторът се инициализира с начална стойност, извлечена спрямо текущото системно време. В документацията пише още, че ако създадем две инстанции на класа **`Random`** в много кратък интервал от време, те най-вероятно ще генерират еднакви числа. Оказва се, че проблемът е в неправилното използване на класа **`Random`**.

Имайки предвид описания проблем, бихме могли да коригираме проблема като създадем инстанция на класа **`Random`** само веднъж при стартиране на програмата. След това при нужда от случайно число ще използваме вече създадения генератор на псевдослучайни числа. Ето как изглежда корекцията в кода:

```
class CardsShuffle
{
  ...

  static Random rand = new Random();

  static void PerformSingleExchange(List<Card> cards)
  {
    int randomIndex = rand.Next(1, cards.Count);
    Card firstCard = cards[0];
    Card randomCard = cards[randomIndex];
    cards[0] = randomCard;
    cards[randomIndex] = firstCard;
  }

  ...
}
```

Изглежда програмата най-сетне работи коректно – при всяко стартиране извежда различна подредба на картите:

```
Initial deck:  (7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6
DIAMOND)(J CLUB)
After shuffle: (2 CLUB)(A SPADE)(J CLUB)(10 DIAMOND)(7 HEART)(6
DIAMOND)
--------------------------------------------------------
Initial deck:  (7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6
DIAMOND)(J CLUB)
After shuffle: (6 DIAMOND)(10 DIAMOND)(J CLUB)(2 CLUB)(A
SPADE)(7 HEART)
```

Пробваме още няколко примера и виждаме, че работи правилно и за тях. Едва сега можем да кажем, че имаме коректна имплементация на алгоритъма, който измислихме в началото и тествахме на хартия.

Стъпка 5 – Вход от конзолата

Остава да реализираме вход от конзолата, за да дадем възможност на потребителя да въведе картите, които да бъдат разбъркани. Забележете, че оставихме за накрая тази стъпка. Защо? Ами много просто: нали не искаме всеки път при стартиране на програмата да въвеждаме 6 карти само за да тестваме дали някаква малка част от кода работи коректно (преди цялата програма да е написана докрай)? Като кодираме твърдо входните данни си спестяваме много време за въвеждането им по време на разработка.

Ако задачата изисква вход от конзолата, реализирайте го задължително най-накрая, след като всичко останало работи. Докато пишете програмата, тествайте с твърдо кодирани примерни данни, за да не въвеждате входа всеки път. Така ще спестите много време и нерви.

Въвеждането на входните данни е хамалска задача, която всеки може да реализира. Трябва само да се помисли в какъв формат се въвеждат картите, дали се въвеждат една по една или всички на един път и дали лицето и боята се задават наведнъж или поотделно. В това няма нищо сложно, така че ще го оставим за упражнение на читателите.

Сортиране на числа – стъпка по стъпка

До момента ви показахме колко важно е да пишете програмата си стъпка по стъпка и преди да преминете на следващата стъпка да се убедите, че предходната е реализирана качествено и работи коректно.

Да разгледаме и задачата със сортиране на числа в нарастващ ред. При нея нещата не стоят по-различно. Отново правилният подход към имплементацията изисква да работим на стъпки. Нека видим накратко кои са стъпките. Няма да пишем кода, но ще набележим основните моменти, през които трябва да преминем. Да предположим, че реализираме идеята за сортиране чрез `List<int>`, в който последователно намираме най-малкото число, отпечатваме го и го изтриваме от списъка с числа. Ето какви биха могли да са стъпките:

> **Стъпка 1.** Измисляме подходящ пример, с който ще си тестваме, например числата 7, 2, 4, 1, 8, 2. Създаваме `List<int>` и го запълваме с числата от нашия пример. Реализираме отпечатване на числата.
>
> Стартираме програмата и тестваме.
>
> **Стъпка 2.** Реализираме метод, който намира най-малкото число в масива и връща позицията му.
>
> Тестваме метода за търсене на най-малко число. Пробваме различни поредици от числа, за да се убедим, че търсенето работи коректно (слагаме най-малкия елемент в началото, в края, в средата; пробваме и когато най-малкия елемент се повтаря няколко пъти).
>
> **Стъпка 3.** Реализираме метод, който намира най-малкото число, отпечатва го и го изтрива.
>
> Тестваме с нашия пример дали методът работи коректно. Пробваме и други примери.
>
> **Стъпка 4.** Реализираме метода, който сортира числата. Той изпълнява предходния метод N пъти (където N е броят на числата).
>
> Задължително тестваме дали всичко работи както трябва.
>
> **Стъпка 5.** Ако е необходим вход от конзолата, реализираме го най-накрая, когато всичко е тествано и работи.

Виждате, че подходът с разбиването на стъпки е приложим при всякакви задачи. Просто трябва да съобразим кои са нашите елементарни стъпки при имплементацията и да ги изпълняваме една след друга, като не забравяме да тестваме всяко парче код възможно най-рано. След всяка стъпка е препоръчително да стартираме програмата, за да се убедим, че до този момент всичко работи правилно. Така ще откриваме евентуални проблеми още при възникването им и ще ги отстраняваме бързо и лесно, а не когато сме написали стотици редове код.

Тествайте решението си!

Това звучи ли ви познато: "Аз съм готов с първа задача. Веднага трябва да започна следващата."? На всеки му е хрумвала такава мисъл, когато е бил на изпит. В програмирането обаче, тази мисъл означава следното:

1. Аз съм разбрал добре условието на задачата.
2. Аз съм измислил алгоритъм за решаването на задачата.
3. Аз съм тествал на лист хартия моя алгоритъм и съм се уверил, че е правилен.
4. Аз съм помислил за структурите от данни и за ефективността на моя алгоритъм.
5. Аз съм написал програма, която реализира коректно моя алгоритъм.
6. Аз съм тествал обстойно моята програма с подходящи примери, за да се уверя, че работи коректно, дори в необичайни ситуации.

Неопитните програмисти почти винаги пропускат последната точка. Те смятат, че тестването не е тяхна задача, което е най-голямата им грешка. Все едно да смятаме, че Майкрософт не са длъжни да тестват Windows и могат да оставят той да "гърми" при всяко второ натискане на мишката.

> ⚠ **Тестването е неразделна част от програмирането! Да пишеш код, без да го тестваш, е като да пишеш на клавиатурата без да виждаш екрана на компютъра – мислиш си, че пишеш правилно, но най-вероятно правиш много грешки.**

Опитните програмисти знаят, че ако напишат код и той не е тестван, това означава, че той още не е завършен. В повечето софтуерни фирми е недопустимо да се предаде код, който не е тестван.

В софтуерната индустрия дори е възприета концепцията за **unit testing** – автоматизирано тестване на отделните единици от кода (методи, класове и цели модули). Unit testing означава за всяка програма да пишем и още една програма, която я тества дали работи коректно. В някои фирми дори първо се измислят тестовите сценарии, пишат се тестовете за програмата и най-накрая се пише самата програма. Темата за unit testing е много сериозна и обемна, но с нея ще се запознаете по-късно, когато навлезете в дълбините на професията "софтуерен инженер". Засега, нека се фокусираме върху ръчното тестване, което всеки един програмист може да извърши, за да се убеди, че неговата програма работи коректно.

Как да тестваме?

Една програма е коректна, ако работи коректно за всеки възможен валиден набор от входни данни. Тестването е процес, който цели да установи наличие на дефекти в програмата, ако има такива. То не може да установи със сигурност дали една програма е коректна, но може да провери с голяма степен на увереност дали в програмата има дефекти, които причиняват некоректни резултати или други проблеми.

За съжаление всички възможни набори входни данни за една програма обикновено са неизброимо много и не може да се тества всеки от тях. Затова в практиката на софтуерното тестване се подготвят и изпълняват такива набори от входни данни (тестове), които целят да обхванат максимално пълно всички различни ситуации (случаи на употреба), които възникват при изпълнение на програмата. Този набор има за цел с минимални усилия (т. е. с минимален брой и максимална простота на тестовете) да провери всички основни случаи на употреба. Ако при тестването по този начин не бъдат открити дефекти, това не доказва, че програмата е 100% коректна, но намалява в много голяма степен вероятността на по-късен етап да се наблюдават дефекти и други проблеми.

Тестването може да установи само наличие на дефекти. То не може да докаже, че дадена програма е коректна! Програмите, които са тествани старателно имат много по-малко дефекти, отколкото програмите, които изобщо не са тествани или не са тествани качествено.

Тестването е добре да започва от един пример, с който обхващаме типичния случай в нашата задача. Той най-често е същият пример, който сме тествали на хартия и за който очакваме нашият алгоритъм да работи коректно.

След написване на кода обикновено следва отстраняване на поредица от дребни грешки и най-накрая нашият пример тръгва. След това е нормално да тестваме програмата с по-голям и по-сложен пример, за да видим как се държи тя в по-сложни ситуации. Следва тестване на граничните случаи и тестване за бързодействие. В зависимост от сложността на конкретната задача могат да се изпълнят от един-два до няколко десетки теста, за да се покрият всички основни случаи на употреба.

При сложен софтуер, например продуктът Microsoft Word броят на тестовете би могъл да бъде няколко десетки, дори няколко стотици хиляди. Ако някоя функция на програмата не е старателно тествана, не може да се твърди, че е реализирана коректно и че работи. Тестването при разработката на софтуер е не по-малко важно от писането на кода. В сериозните софтуерни корпорации на един програмист се пада поне един тестер. Например в Microsoft на един програмист, който пише код (software engineer) се назначават средно по двама души, които тестват кода (software quality assurance engineers). Тези разработчици също са програмисти, но не пишат основния софтуер, а пишат тестващи програми за него, които позволяват цялостно автоматизирано тестване.

Тестване с добър представител на общия случай

Както вече споменахме, нормално е тестването да започне с тестов пример, който е добър представител на общия случай. Това е тест, който хем е достатъчно прост, за да бъде проигран ръчно на хартия, хем е достатъчно общ, за да покрие общия случай на употреба на програмата, а не някой частен случай. Следвайки този подход най-естественото нещо, което някой програмист може да направи е следното:

1. Да измисли пример, който е добър представител на общия случай.
2. Да тества примера на ръка (на хартия).
3. Да очаква примера да тръгне успешно и от имплементацията на неговия алгоритъм.
4. Да се убеди, че примерът му работи коректно след написване на програмата и отстраняване на грешките, които възникват при писането на кода.

За съжаление много програмисти спират с тестването в този момент. Някои по-неопитни програмисти правят дори нещо по-лошо: измислят какъв да е пример (който е прост частен случай на задачата), не го тестват на хартия, пишат някакъв код и накрая като тръгне този пример, решават, че са приключили. Не правете така! Това е като да ремонтираш лека кола и когато си готов, без да запалиш двигателя да пуснеш колата леко по някой наклон и ако случайно тръгне надолу да се произнесеш компетентно и безотговорно: "Готова е колата. Ето, движи се надолу без никакъв проблем."

Какво още да тестваме?

Тестването на примера, който сте проиграли на хартия е едва първата стъпка от тестването на програмата. Следва да извършите още няколко задължителни теста, с които да се убедите, че програмата ви работи коректно:

- Сериозен тест за обичайния случай. Целта на този тест е да провери дали за по-голям и по-сложен пример вашата програма работи коректно. За нашата задача с разбъркването на картите такъв тест може да е тесте от 52 карти.
- Тестове за граничните случаи. Те проверяват дали вашата програма работи коректно при необичаен вход на границата на допустимото. За нашата задача такъв пример е разбъркването на тесте, което се състои само от една карта.
- Тестове за бързодействие. Тези тестове поставят програмата в екстремални условия като ѝ подават големи по размерност входни данни и проверяват бързодействието.

Нека разгледаме горните групи тестове една по една.

Сериозен тест на обичайния случай

Вече сме тествали програмата за един случай, който сме измислили на ръка и сме проиграли на хартия. Тя работи коректно. Този случай покрива типичния сценарий за употреба на програмата. Какво повече трябва да тестваме? Ами много просто, възможно е програмата да е грешна, но да работи по случайност за нашия случай.

Как да подготвим по-сериозен тест? Това зависи много от самата задача. Тестът хем трябва да е с по-голям обем данни, отколкото ръчния тест, но все пак трябва да можем да проверим дали изхода от програмата е коректен.

За нашия пример с разбъркването на карти в случаен ред е нормално да тестваме с пълно тесте от 52 карти. Лесно можем да произведем такъв входен тест с два вложени цикъла. След изпълнение на програмата също лесно можем да проверим дали резултатът е коректен – трябва картите да са разбъркани и разбъркването да е случайно. Необходимо е още при две последователни изпълнения на този тест да се получи тотално различно разбъркване. Ето как изглежда кодът, реализиращ такъв тест:

```
static void TestShuffle52Cards()
{
    List<Card> cards = new List<Card>();
    string[] allFaces = new string[] { "2", "3", "4", "5",
        "6", "7", "8", "9", "10", "J", "Q", "K", "A" };
    Suit[] allSuits = new Suit[] { Suit.CLUB, Suit.DIAMOND,
        Suit.HEART, Suit.SPADE };
    foreach (string face in allFaces)
    {
        foreach (Suit suit in allSuits)
        {
            Card card = new Card() { Face = face, Suit = suit };
            cards.Add(card);
        }
    }
    ShuffleCards(cards);
    PrintCards(cards);
}
```

Ако го изпълним получаваме примерно такъв резултат:

```
(4 DIAMOND)(2 DIAMOND)(6 HEART)(2 SPADE)(A SPADE)(7 SPADE)(3
DIAMOND)(3 SPADE)(4 SPADE)(4 HEART)(6 CLUB)(K HEART)(5 CLUB)(5
DIAMOND)(5 HEART)(A HEART)(9 CLUB)(10 CLUB)(A CLUB)(6 SPADE)(7
CLUB)(7 DIAMOND)(3 CLUB)(9 HEART)(8 CLUB)(3 HEART)(9 SPADE)(4
CLUB)(8 HEART)(9 DIAMOND)(5 SPADE)(8 DIAMOND)(J HEART)(10
DIAMOND)(10 HEART)(10 SPADE)(Q HEART)(2 CLUB)(J CLUB)(J SPADE)(Q
CLUB)(7 HEART)(2 HEART)(Q SPADE)(K CLUB)(J DIAMOND)(6 DIAMOND)(K
SPADE)(8 SPADE)(A DIAMOND)(Q DIAMOND)(K DIAMOND)
```

Ако огледаме внимателно получената подредба на картите, ще забележим, че много голяма част от тях си стоят на първоначалната позиция и не са променили местоположението си. Например сред първите 4 карти половината не са били разместени при разбъркването: 2♦ и 2♠.

Никога не е късно да намерим дефект в програмата и единственият начин да направим това е да тестваме сериозно, задълбочено и систематично кода с примери, които покриват най-разнообразни практически ситуации. Полезно беше да направим тест с реално тесте от 52 карти, нали? Натъкнахме се на сериозен дефект, който не може да бъде подминат.

Сега как да оправим проблема? Първата идея, която ни хрумва, е да правим по-голям брой случайни единични размествания (очевидно N на брой са недостатъчни). Друга идея е N-тото разместване да разменя N-тата поред карта от тестето със случайна друга карта, а не винаги първата. Така ще си гарантираме, че всяка карта ще бъде разменена с поне една друга карта и няма да останат позиции от тестето, които не са участвали в нито една размяна (това се наблюдава в горния пример с разбъркването на 52 карти). Втората идея изглежда по-надеждна. Нека я имплементираме. Получаваме следните промени в кода:

```
static void PerformSingleExchange(List<Card> cards, int index)
{
  int randomIndex = rand.Next(1, cards.Count);
  Card firstCard = cards[index];
  Card randomCard = cards[randomIndex];
  cards[index] = randomCard;
  cards[randomIndex] = firstCard;
}

static void ShuffleCards(List<Card> cards)
{
  for (int i = 0; i < cards.Count; i++)
  {
    PerformSingleExchange(cards, i);
  }
}
```

Стартираме програмата и получаваме много по-добро разбъркване на тестето от 52 карти, отколкото преди:

```
(9 HEART)(5 CLUB)(3 CLUB)(7 SPADE)(6 CLUB)(5 SPADE)(6 HEART)(4
CLUB)(10 CLUB)(3 SPADE)(K DIAMOND)(10 HEART)(8 CLUB)(A CLUB)(J
DIAMOND)(K SPADE)(9 SPADE)(7 CLUB)(10 DIAMOND)(9 DIAMOND)(8
HEART)(6 DIAMOND)(8 SPADE)(5 DIAMOND)(4 HEART)(10 SPADE)(J
CLUB)(Q SPADE)(9 CLUB)(J HEART)(K CLUB)(2 HEART)(7 HEART)(A
HEART)(3 DIAMOND)(K HEART)(A SPADE)(8 DIAMOND)(4 SPADE)(3
HEART)(5 HEART)(Q HEART)(4 DIAMOND)(2 SPADE)(A DIAMOND)(2
DIAMOND)(J SPADE)(7 DIAMOND)(Q DIAMOND)(2 CLUB)(6 SPADE)(Q
CLUB)
```

Изглежда, че най-сетне картите са подредени случайно и са различни при всяко изпълнение на програмата. Няма видими дефекти (например повтарящи се или липсващи карти или карти, които често запазват началната си позиция). Програмата работи бързо и не зависва. Изглежда сме се справили добре.

Нека вземем другата примерна задача: сортиране на числа. Как да си направим сериозен тест за обичайния случай? Ами най-лесното е да генерираме поредица от 100 или дори 1000 случайни числа и да ги сортираме. Проверката за коректност е лесна: трябва числата да са подредени по големина. Друг тест, който е удачен при сортирането на числа е да вземем числата от 1000 до 1 в намаляващ ред и да ги сортираме. Трябва да получим същите числа, но сортирани в нарастващ ред. Би могло да се каже, че това е най-трудният възможен тест за тази задача и ако той работи за голям брой числа, значи програмата най-вероятно е коректна.

Нека разгледаме и другите видове тестове, които е добре винаги да правим при решението на задачи по програмиране.

Гранични случаи

Най-честото нещо, което се пропуска при решаването на задачи, пък и въобще в програмирането, е да се помисли за граничните ситуации. Граничните ситуации се получават при входни данни на границата на нормалното и допустимото. При тях често пъти програмата гърми, защото не очаква толкова малки или големи или необичайни данни, но те все пак са допустими по условие или не са допустими, но не са предвидени от програмиста.

Как да тестваме граничните ситуации? Ами разглеждаме всички входни данни, които програмата получава и се замисляме какви са екстремните им стойности и дали са допустими. Възможно е да имаме екстремно малки стойности, екстремно големи стойности или просто странни комбинации от стойности. Ако по условие имаме ограничения, например до 52 карти, стойностите около това число 52 също са гранични и могат да причинят проблеми.

Граничен случай: разбъркване на една карта

Например в нашата задача за разбъркване на карти граничен случай е да разбъркаме една карта. Това е съвсем валидна ситуация (макар и необичайна), но нашата програма би могла да не работи коректно за една карта поради някакви особености. Нека проверим какво става при разбъркване на една карта. Можем да напишем следния малък тест:

```
static void TestShuffleOneCard()
{
   List<Card> cards = new List<Card>();
   cards.Add(new Card() { Face = "A", Suit = Suit.CLUB });
   CardsShuffle.ShuffleCards(cards);
   CardsShuffle.PrintCards(cards);
}
```

Изпълняваме го и получаваме напълно неочакван резултат:

```
Unhandled Exception: System.ArgumentOutOfRangeException: Index
was out of range. Must be non-negative and less than the size of
the collection. Parameter name: index
   at
System.ThrowHelper.ThrowArgumentOutOfRangeException(ExceptionArgu
ment argument, ExceptionResource resource)
   at System.ThrowHelper.ThrowArgumentOutOfRangeException()
   at System.Collections.Generic.List`1.get_Item(Int32 index)
   at CardsShuffle.PerformSingleExchange(List`1 cards, Int32
index) in D:\Projects\Cards\CardsShuffle.cs:line 61
…
```

Ясно е какъв е проблемът: генерирането на случайно число се счупи, защото му се подава невалиден диапазон. Нашата програма работи добре при нормален брой карти, но не работи за една карта. Открихме лесен за отстраняване дефект, който бихме пропуснали с лека ръка, ако не бяхме разгледали внимателно граничните случаи. След като знаем какъв е проблемът, поправката на кода е тривиална:

```
static void ShuffleCards(List<Card> cards)
{
   if (cards.Count > 1)
   {
      for (int i = 0; i < cards.Count; i++)
      {
         PerformSingleExchange(cards, i);
      }
   }
}
```

Тестваме отново и се убеждаваме, че проблемът е решен.

Граничен случай: разбъркване на две карти

Щом има проблем за 1 карта, сигурно може да има проблем и за 2 карти. Не звучи ли логично? Нищо не ни пречи да проверим. Стартираме програмата с 2 карти няколко пъти и очакваме да получим различни размествания на двете карти. Ето примерен код, с който можем да направим това:

```
static void TestShuffleTwoCards()
{
    List<Card> cards = new List<Card>();
    cards.Add(new Card() { Face = "A", Suit = Suit.CLUB });
    cards.Add(new Card() { Face = "3", Suit = Suit.DIAMOND });
    CardsShuffle.ShuffleCards(cards);
    CardsShuffle.PrintCards(cards);
}
```

Стартираме няколко пъти и резултатът винаги е все един и същ:

```
(3 DIAMOND)(A CLUB)
```

Изглежда пак нещо не е наред. Ако разгледаме кода или го пуснем през дебъгера, ще се убедим, че всеки път се правят точно два размествания: разменя се първата карта с втората и веднага след това се разменя втората карта с първата. Резултатът винаги е един и същ. Как да решим проблема? Веднага можем да се сетим за няколко решения:

- Правим единичното разместване N+K брой пъти, където K е случайно число между 0 и 1.
- При разместванията допускаме случайната позиция, на която отива първата карта да включва и нулевата позиция.
- Разглеждаме случая с точно 2 карти като специален и пишем отделен метод специално за него.

Второто решение изглежда най-просто за имплементация. Да го пробваме. Получаваме следния код:

```
static void PerformSingleExchange(List<Card> cards, int index)
{
    int randomIndex = rand.Next(0, cards.Count);
    Card firstCard = cards[index];
    Card randomCard = cards[randomIndex];
    cards[index] = randomCard;
    cards[randomIndex] = firstCard;
}
```

Тестваме отново разбъркването на две карти и този път изглежда, че програмата работи коректно: картите се разместват понякога, а понякога запазват началната си подредба.

Щом има проблем за 2 карти, може да има проблем и за 3 карти, нали? Ако тестваме програмата за 3 карти, ще се убедим, че тя работи коректно.

След няколко стартирания получаваме всички възможни разбърквания на трите карти, което показва, че случайното разбъркване може да получи всички пермутации на трите карти. Този път не открихме дефекти и програмата няма нужда от промяна.

Граничен случай: разбъркване на нула карти

Какво още може да проверим? Има ли други необичайни, гранични ситуации. Да помислим. Какво ще стане, ако се опитаме да разбъркаме празен списък от карти? Това наистина е малко странно, но има едно правило, че една програма трябва или да работи коректно или да сигнализира за грешка. Нека да видим какво ще върне нашата програма за 0 карти. Резултатът е празен списък. Коректен ли е? Ами да, ако разбъркаме 0 карти в случаен ред би трябвало да получим пак 0 карти. Изглежда всичко е наред.

⚠ **При грешни входни данни програмата не трябва да връща грешен резултат, а трябва или да върне верен резултат или да съобщи, че входните данни са грешни.**

Какво мислите за горното правило? Логично е нали? Представете си, че правите програма, която показва графични изображения (снимки). Какво става при снимка, която представлява празен файл. Това е също необичайна ситуация, която не би трябвало да се случва, но може да се случи. Ако при празен файл вашата програма зависва или хвърля необработено изключение, това би било много досадно за потребителя. Нормално е празният файл да бъде изобразен със специална икона или вместо него да се изведе съобщение "Invalid image file", нали?

Помислете колко гранични и необичайни ситуации има в Windows. Какво става ако печатаме празен файл на принтера? Дали Windows забива в този момент и показва небезизвестния "син екран"? Какво става, ако в калкулаторът на Windows направим деление на нула? Какво става, ако копираме празен файл (с дължина 0 байта) с Windows Explorer? Какво става, ако в Notepad се опитаме да създадем файл без име (с празен низ, зададен като име)? Виждате, че гранични ситуации има много и навсякъде. Наша задача като програмисти е да ги улавяме и да мислим за тях преди още да се случат, а не едва когато неприятно развълнуван потребител яростно ни нападне по телефона с неприлични думи по адрес на наши близки роднини.

Да се върнем на нашата задача за разбъркване на картите. Оглеждайки се за гранични и необичайни случаи се сещаме дали можем да разбъркаме -1 карти? Понеже няма как да създадем масив с -1 елемента, считаме, че такъв случай няма как да се получи.

Понеже нямаме горна граница на картите, няма друга специална точка (подобна на ситуацията с 1 карта), около която да търсим за проблемни ситуации. Прекратяваме търсенето на гранични случаи около броя на картите. Изглежда предвидихме всички ситуации.

Остава да се огледаме дали няма други стойности от входните данни, които могат да причинят проблеми, например невалидна карта, карта с невалидна боя, карта с отрицателно лице (примерно -1 спатия) и т.н. Като се замислим, нашият алгоритъм не се интересува какво точно разбърква (карти за игра или яйца за омлет), така че това не би трябвало да е проблем. Ако имаме съмнения, можем на си направим тест и да се убедим, че при невалидни карти резултатът от разбъркването им не е грешен.

Оглеждаме се за други гранични ситуации във входните данни и не се сещаме за такива. Остава единствено да измерим бързодействието, нали? Всъщност пропуснахме нещо много важно: да тестваме всичко наново след поправките.

Повторно тестване след корекциите (regression testing)

Често пъти при корекции на грешки се получават незабелязано нови грешки, които преди не са съществували. Например, ако поправим грешката за 2 карти чрез промяна на правилата за размяна на единична карта, това би могло да доведе до грешен резултат при 3 или повече карти. При всяка промяна, която би могла да засегне други случаи на употреба, е задължително да изпълняваме отново тестовете, които сме правили до момента, за да сме сигурни, че промяната не поврежда вече работещите случаи. За тази цел е добре да запазваме тестовете на програмата, които сме изпълнявали, като методи (например започващи с префикс `Test`), а не да ги изтриваме.

Идеята за повторяемост на тестовете лежи в основата на концепцията unit testing. Тази тема, както вече споменахме, е за по-напреднали и затова я оставяме за по-нататък във времето (и пространството).

В нашия случай с разбъркването на карти след всички промени, които направихме, е редно да тестваме отново разбъркването на 0 карти, на 1 карта, на 2 карти, на 3 карти и на 52 карти.

Когато сте открили и сте поправили грешка в кода, отнасяща се за някой специфичен тест, уверете се, че поправката не засяга всички останали тестове. За целта е препоръчително да запазвате всички тестове, които изпълнявате.

Тестове за производителност

Нормално е винаги, когато пишете софтуер, да имате някакви изисквания и критерии за бързодействие на програмите или модулите, които пишете.

Никой не обича машината му да работи бавно, нали? Затова трябва да се стремите да не пишете софтуер, който работи бавно, освен, ако нямате добра причина за това.

Как тестваме бързодействието (производителността) на програмата? Първият въпрос, който трябва да си зададем, когато стигнем до тестване на бързодействието, е имаме ли изисквания за скорост. Ако имаме, какви са те? Ако нямаме, какви ориентировъчни критерии за бързодействие трябва да спазим (винаги има някакви общоприети)?

Разбъркване на карти – тестове за производителност

Нека да разгледаме за пример нашата програма за разбъркване на тесте карти. Какви изисквания за бързодействие би могла да има тя? Първо имаме ли по условие такива изисквания? Нямаме изрично изискване в стил "програмата трябва да завършва за една секунда или по-бързо при 500 карти на съвременна компютърна конфигурация". Щом нямаме такива изрични изисквания, все пак трябва някак да решим въпроса с оценката на бързодействието, неформално, по усет.

Понеже работим с карти за игра, считаме, че едно тесте има 52 карти. Вече пускахме такъв тест и видяхме, че работи мигновено, т.е. няма видимо забавяне. Изглежда за нормалния случай на употреба бързината не създава проблеми.

Нормално е да тестваме програмата и с много повече карти, примерно с 52 000, защото в някой специален случай някой може да реши да разбърква много карти и да срещне проблеми. Лесно можем да си направим такъв пример като добавим 1 000 пъти нашите 52 карти и след това ги разбъркаме. Нека пуснем един такъв пример:

```
static void TestShuffle52000Cards()
{
  List<Card> cards = new List<Card>();
  string[] allFaces = new string[] {"2", "3", "4", "5",
    "6", "7", "8", "9", "10", "J", "Q", "K", "A"};
  Suit[] allSuits = new Suit[] { Suit.CLUB, Suit.DIAMOND,
    Suit.HEART, Suit.SPADE};
  for (int i = 0; i < 1000; i++)
  {
    foreach (string face in allFaces)
    {
      foreach (Suit suit in allSuits)
      {
        Card card = new Card() { Face = face, Suit = suit };
        cards.Add(card);
      }
    }
  }
  ShuffleCards(cards);
```

```
   PrintCards(cards);
}
```

Стартираме програмата и забелязваме, че машината леко се успива за около пет-десет секунди. Разбира се при по-бавни машини успиването е за по-дълго. Какво се случва? Би трябвало при 52 000 карти да направим приблизително същия брой единични размествания, а това би трябвало да отнеме частица от секундата. Защо имаме секунди забавяне? Опитните програмисти веднага ще се сетят, че печатаме големи обеми информация на конзолата, а това е бавна операция. Ако коментираме реда, в който отпечатваме резултата и измерим времето за изпълнение на разбъркването на картите, ще се убедим, че програмата работи достатъчно бързо дори и за 52 000 карти. Ето как можем да замерим времето:

```
static void TestShuffle52000Cards()
{
   ...
   DateTime oldTime = DateTime.Now;
   ShuffleCards(cards);
   DateTime newTime = DateTime.Now;
   Console.WriteLine("Execution time: {0}", newTime - oldTime);
   //PrintCards(cards);
}
```

Можем да проверим точно колко време отнема изпълнението на метода за разбъркване на картите:

```
Execution time: 00:00:00.0156250
```

Една милисекунда изглежда напълно приемливо. Нямаме проблем с бързодействието.

Сортиране на числа – тестове за производителност

Нека разгледаме другата от нашите примерни задачи: сортиране на масив с числа. При нея бързодействието може да се окаже много по-проблемно, отколкото разбъркването на тесте карти. Нека сме направили просто решение, което работи така: намира най-малкото число в масива и го разменя с числото на позиция 0. След това намира сред останалите числа най-малкото и го поставя на позиция 1. Това се повтаря докато се стигне до последното число, което би трябвало да си е вече на мястото. Няма да коментираме верността на този алгоритъм. Той е добре известен като "метод на пряката селекция" (http://en.wikipedia.org/wiki/Selection_sort).

Сега да предположим, че сме минали през всички стъпки за решаването на задачи по програмиране и накрая сме стигнали до този пример, с който се опитваме да сортираме 10 000 случайни числа:

Sort10000Numbers.cs

```
using System;

public class Sort10000Numbers
{
   static void Main()
   {
      int[] numbers = new int[10000];
      Random rnd = new Random();
      for (int i=0; i<numbers.Length; i++)
      {
         numbers[i] = rnd.Next(2 * numbers.Length);
      }
      SortNumbers(numbers);
      PrintNumbers(numbers);
   }

   static void SortNumbers(int[] numbers)
   {
      for (int i = 0; i < numbers.Length - 1; i++)
      {
         int minIndex = i;
         for (int j=i+1; j<numbers.Length; j++)
         {
            if (numbers[j] < numbers[minIndex])
            {
               minIndex = j;
            }
         }
         int oldNumber = numbers[i];
         numbers[i] = numbers[minIndex];
         numbers[minIndex] = oldNumber;
      }
   }

   static void PrintNumbers(int[] numbers)
   {
      Console.Write("[");
      for (int i = 0; i < numbers.Length; i++)
      {
         Console.Write(numbers[i]);
         if (i < numbers.Length - 1)
         {
            Console.Write(", ");
         }
      }
      Console.WriteLine("]");
   }
}
```

Стартираме го и изглежда, че той работи за под секунда на нормална съвременна машина. Резултатът (със съкращения) би могъл да е нещо такова:

```
[0, 14, 19, 20, 20, 22, ..., 19990, 19993, 19995, 19996]
```

Сега правим още един експеримент за 300 000 случайни числа и виждаме, че програмата като че ли зависва или работи прекалено бавно, за да я изчакаме. Това е сериозен проблем с бързодействието.

Преди да се втурнем да го решаваме трябва, обаче, да си зададем един много важен въпрос: дали ще имаме реална ситуация, при която ще се наложи да сортираме 300 000 числа. Ако сортираме примерно оценките на студентите в един курс, те не могат да бъдат повече от няколко десетки. Ако, обаче, сортираме цените на акциите на голяма софтуерна компания за цялата й история на съществуване на фондовата борса, можем да имаме огромен брой числа, защото цената на акциите й може да се променя всяка секунда. За десетина години цените на акциите на тази компания биха могли да се променят няколкостотин милиона пъти. В такъв случай трябва да търсим по-ефективен алгоритъм за сортиране.

Как да правим ефективно сортиране на цели числа можем да прочетем в десетки сайтове в Интернет и в класическите книги по алгоритми. Конкретно за тази задача най-подходящо е да използваме алгоритъма за сортиране "radix sort" (http://en.wikipedia.org/wiki/Radix_sort), но тази дискусия е извън темата и ще я пропуснем.

Нека припомним доброто старо правило за ефективността:

⚠ **Винаги трябва да правим компромис между времето, което ще вложим, за да напишем програмата, и бързодействието, което искаме да постигнем. Иначе може да изгубим време да решаваме проблем, който не съществува или да стигнем до решение, което не върши работа.**

Трябва да имаме предвид и че за някои задачи изобщо не съществуват бързи алгоритми и ще трябва да се примирим с ниската производителност. Например за задачата за намиране на всички прости делители на цяло число (вж. http://en.wikipedia.org/wiki/Integer_factorization) няма известно бързо решение.

За някои задачи нямаме нужда от бързина, защото очакваме входните данни да са достатъчно малки и тогава е безумно да търсим сложни алгоритми с цел бързодействие. Например задачата за сортиране на оценките на студентите от даден курс може да се реши с произволен алгоритъм за сортиране и при всички случаи ще работи бързо, тъй като броят на студентите се очаква да е достатъчно малък.

Генерални изводи

Преди да започнете да четете настоящата тема сигурно сте си мислили, че това ще е най-скучната и безсмислена до момента, но вярваме, че сега мислите по съвсем различен начин. Всички си мислят, че знаят как да решават задачи по програмиране и че за това няма "рецепта" (просто трябва да го можеш), но въобще не е така. Има си рецепти и то най-различни. Ние ви показахме нашата и то в действие! Убедихте се, че нашата рецепта дава резултат, нали?

Само се замислете колко грешки и проблеми открихме докато решавахме една много лесна и проста задача: разбъркване на карти в случаен ред. Щяхме ли да напишем качествено решение, ако не бяхме подходили към задачата по рецептата, изложена по-горе? А какво би се случило, ако решаваме някоя много по-сложна и трудна задача, примерно да намерим оптимален път през сутрешните задръствания в София по карта на града с актуални данни за трафика? При такива задачи е абсолютно немислимо да подходим хазартно и да се хвърлим на първата идея, която ни дойде на ум. Първата стъпка към придобиване на умения за решаване на такива сложни задачи е да се научите да подхождате към задачата систематично и да усвоите рецептата за решаване на задачи, която ви демонстрирахме в действие. Това, разбира се, съвсем няма да ви е достатъчно, но е важна крачка напред!

За решаването на задачи по програмиране си има рецепта! Ползвайте систематичен подход и ще имате много по-голям успех, отколкото, ако карате по усет. Дори професионалистите с десетки години опит ползват в голяма степен описания от нас подход. Ползвайте го и вие и ще се убедите, че работи! Не пропускайте да тествате сериозно и задълбочено решението.

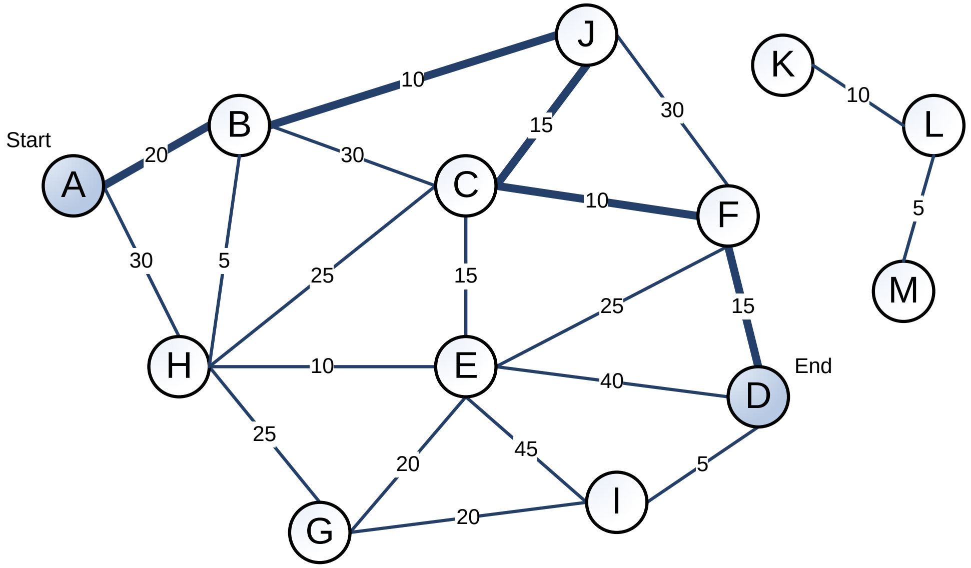
Упражнения

1. Използвайки описаната в тази глава методология за решаване на задачи по програмиране решете следната задача: дадени са N точки (N < 100 000) в равнината. Точките са представени с целочислени координати (x_i, y_i). Напишете програма, която намира всички възможни хоризонтални или вертикални прави (отново с целочислени координати), които разделят равнината на две части, така че двете части да съдържат по равен брой точки (точките попадащи върху линията не се броят).
2. Използвайки описаната в тази глава методология за решаване на задачи по програмиране решете следната задача: дадено е множество

S от n цели числа и положително цяло число **`k`** **`(k <= n <= 10)`**. Алтернативна редица от числа е редица, която алтернативно сменя поведението си от растяща към намаляваща и обратно след всеки неин елемент. Напишете програма, която генерира всички алтернативни редици **`s`**$_1$**`,`** **`s`**$_2$**`, …,`** **`s`**$_k$ състояща се от **`k`** различни елемента от S. Пример: **`S=`**{ 2, 5, 3, 4 }, **`K=3`**: {2, 4, 3}, {2, 5, 3}, {2, 5, 4}, {3, 2, 4}, {3, 2, 5}, {3, 4, 2}, {3, 5, 2}, {3, 5, 4}, {4, 2, 3}, {4, 2, 5}, {4, 3, 5}, {5, 2, 3}, {5, 2, 4}, {5, 3, 4}

3. Използвайки описаната в тази глава методология за решаване на задачи по програмиране решете следната задача: разполагаме с карта на един град. Картата се състои от улици и кръстовища. За всяка улица на картата е отбелязана нейната дължина. Едно кръстовище свързва няколко улици. Задачата е да се намери и отпечата най-късият път между двойка кръстовища (измерен като суми от дължините на улиците, през които се преминава).

   Ето как изглежда схематично картата на един примерен град:



   На тази карта най-късият път между кръстовища A и D е с дължина 70 и е показан на фигурата с удебелени линии. Както виждате, между A и D има много пътища с най-различна дължина. Не винаги най-късото начало води към най-късия път и не винаги най-малкият брой улици води до най-къс път. Между някои двойки кръстовища дори въобще не съществува път. Това прави задачата доста интересна.

   Входните данни се задават в текстов файл **`map.txt`**. Файлът започва със списък от улици и техните дължини, след което следва празен ред и след него следват двойки кръстовища, между които се търси най-кратък път. Файлът завършва с празен ред. Ето пример:

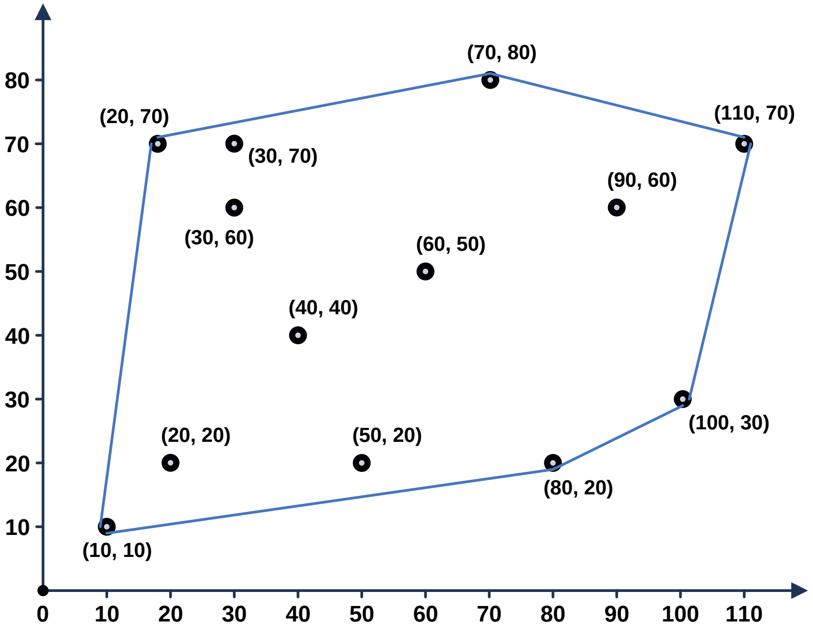
```
A B 20
A H 30
B H 5
...
L M 5
(празен ред)
A D
H K
A E
(празен ред)
```

Резултатът от изпълнението на програмата за всяка двойка кръстовища от списъка във входния файл трябва да е дължината на най-късия път, следвана от самия път. За картата от нашия пример изходът трябва да изглежда така:

```
70 ABJCFD
No path!
35 ABHE
```

4. * В равнината са дадени са N точки с координати цели, положителни числа. Тези точки представляват дръвчета в една нива. Стопанинът на нивата иска да огради дръвчетата, като използва минимално количество ограда. Напишете програма, която намира през кои точки трябва да минава оградата. Използвайте методологията за решаване на задачи по програмиране!

   Ето как би могла да изглежда градината:



Входните данни се четат от файл **`garden.txt`**. На първия ред на файла е зададен броят на точките. Следват координатите на точките. За нашия пример входният файл би могъл да има следното съдържание:

```
13
60 50
100 30
40 40
20 70
50 20
30 70
10 10
110 70
90 60
80 20
70 80
20 20
30 60
```

Изходните данни трябва да се отпечатат на конзолата в като последователност от точки, през които оградата трябва да мине. Ето примерен изход:

```
(10, 10) - (20, 70) - (70, 80) - (110, 70) - (100, 30) - (80,
20) - (10, 10)
```

Решения и упътвания

1. Първо намерете подходящ кариран лист хартия, на който да можете да си разчертаете координатната система и точките върху нея докато разписвате примери и мислите за конкретно решение. Помислете за различни решения и сравнете кое от тях е по-добро от гледна точка на качество, бързина на работа и време необходимо за реализацията.

2. Отново хванете първо лист и химикалка. Разпишете много примери и помислете върху тях. Какви идеи ви идват? Нужни ли са ви още примери за да се сетите за решение? Обмислете идеи, разпишете ги първо на хартия, ако сте сигурни в тях ги реализирайте. Помислете за примери, върху които вашата програма няма да работи коректно. Винаги е добра идея първо да измислите особените примери, чак след това да реализирате решението си. Помислете как ще работи вашето решение при различни стойности на **`K`** и различни стойности и брой елементи в **`S`**.

3. Следвайте стриктно методологията за решаване на задачи по програмиране! Задачата е сложна и изисква да й отделите достатъчно внимание. Първо си нарисувайте примера на хартия. Опитайте се да измислите сами правилен алгоритъм за намиране на най-къс път. След

това потърсете в Интернет по ключови думи "shortest path algorithm". Много е вероятно бързо да намерите статия с описание на алгоритъм за най-къс път.

Проверете дали алгоритъмът е верен. Пробвайте различни примери.

В каква структура от данни ще пазите картата на града? Помислете кои са операциите, които ви трябват в алгоритъма за най-къс път. Вероятно ще стигнете до идеята да пазите списък от улиците за всяко кръстовище, а кръстовищата да пазите в списък или хеш-таблица.

Помислете за ефективността. Ще работи ли вашият алгоритъм за 1 000 кръстовища и 5 000 улици?

Пишете стъпка по стъпка. Първо направете четенето на входните данни. (В случая входът е от файл и затова можете да започнете от него. Ако входът беше от конзолата, трябваше да го оставите за най-накрая). Реализирайте отпечатване на прочетените данни. Реализирайте алгоритъма за най-къс път. Ако можете, разбийте реализацията на стъпки. Например като за начало можете да търсите само дължината на най-късия път без самия път (като списък от кръстовища), защото е по-лесно. Реализирайте след това и намирането на самия най-къс път. Помислете какво става, ако има няколко най-къси пътя с еднаква дължина. Накрая реализирайте отпечатването на резултатите, както се изисква в условието на задачата.

Тествайте решението си! Пробвайте с празна карта. Пробвайте с карта с 1 кръстовище. Пробвайте случай, в който няма път между зададените кръстовища. Пробвайте с голяма карта (1 000 кръстовища и 5 000 улици). Можете да си генерирате такава с няколко реда програмка. За имената на кръстовищата трябва да използвате **`string`**, а не **`char`**, нали? Иначе как ще имате 1 000 кръстовища? Работи ли вашето решение бързо? А работи ли вярно?

Внимавайте с входните и изходните данни. Спазвайте формата, който е указан в условието на задачата!

4. Ако не сте много силни в аналитичната геометрия, едва ли ще измислите решение на задачата сами. Опитайте търсене в Интернет по ключовите думи "convex hull algorithm". Знаейки, че оградата, която трябва да построим се нарича "изпъкнала обвивка" (convex hull) на множество точки в равнината, ще намерим стотици статии в Интернет по темата, в някои дори има сорс код на C#. Не преписвайте грешките на другите и особено сорс кода! Мислете! Проучете как работи алгоритъмът и си го реализирайте сами.

   Проверете дали алгоритъмът е верен. Пробвайте различни примери (първо на хартия). Какво става, ако има няколко точки на една линия върху изпъкналата обвивка? Трябва ли да включвате всяка от тях? Помислете какво става, ако има няколко изпъкнали обвивки. От коя точка започвате? По часовниковата стрелка ли се движите или обратното? В условието на задачата има ли изискване как точно да са подредени точките в резултата?

В каква структура от данни ще пазите точките? В каква структура ще пазите изпъкналата обвивка?

Помислете за ефективността. Ще работи ли вашият алгоритъм за 1 000 точки?

Пишете стъпка по стъпка. Първо направете четенето на входните данни. Реализирайте отпечатване на прочетените точки. Реализирайте алгоритъма за изпъкнала обвивка. Ако можете, разбийте реализацията на стъпки. Накрая реализирайте отпечатването а резултата, както се изисква в условието на задачата.

Тествайте решението си! Какво става, ако имаме 0 точки? Пробвайте с една точка. Пробвайте с 2 точки. Пробвайте с 5 точки, които са на една линия. Работи ли алгоритъмът ви? Какво става, ако имаме 10 точки и още 10, които съвпадат с първите 10? Какво става, ако имаме 10 точки, всичките една върху друга? Какво става, ако имаме много точки, примерно 1 000. Работи ли бързо вашият алгоритъм? Какво става, ако координатите на точките са големи числа, примерно (100 000 000, 200 000 000)? Влияе ли това на вашия алгоритъм? Имате ли грешки от загуба на точност?

Внимавайте с входните и изходните данни. Спазвайте формата, който е указан в условието на задачата! Не си измисляйте сами формата на входния файл и на изхода. Те са ясно дефинирани и трябва да се спазват.

Ако имате мерак, направете си визуализация на точките и изпъкналата обвивка като графика с Windows Forms или WPF. Направете си и генератор на случайни тестови данни и си тествайте многократно решението, като гледате визуализацията на обвивката – дали коректно обвива точките и дали е минимална.

Telerik Academy SCHOOL

Стани част от Академията на Телерик за ученици!

„Основната цел на обучението е да израстнеш, а нашите умове, за разлика от телата ни, могат да растат през целия ни живот!" Мортимър Адлър

АКАДЕМИЯТА НА ТЕЛЕРИК за ученици предоставя съвременно, специализирано обучение по софтуерно инженерство за ученици от средните училища в България. Нашата основна цел е да помогнем на младите момчета и момичета да се подготвят за Националната олимпиада по информационни технологии (НОИТ) и да придобият солидни теоретични и практически умения за разработка на софтуер и софтуерни технологии.

В Училищната академия на Телерик по софтуерно инженерство:

- **Стимулираме креативността и мисленето на учениците**
- **Всеки ученик получава индивидуално внимание и съдействие**
- **Полагаме основите за писане на качествен програмен код**
- **Изучаваме съвременните софтуерни технологии**
- **Подготвяме учениците за Националната олимпиада по информационни технологии**
- **Предлагаме голямо разнообразие от практически и теоретични обучения**

Програмата на Академията включва:

C#, .NET технологии, бази данни, SQL, Silverlight, 3D графики и моделиране, разработка на игри, програмиране на микроконтролери, WPF, ASP.NET, HTML5, разработка на мобилни приложения, основи на софтуерното инженерство, работа в екип

schoolacademy.telerik.com

groups.google.com/group/it-olymp *facebook.com/TelerikSchoolAcademy*

telerik
deliver more than expected

Глава 24. Практически задачи за изпит по програмиране – тема 1

В тази тема...

В настоящата тема ще разгледаме условията и ще предложим решения на три задачи от примерен изпит по програмиране. При решаването им ще приложим на практика описаната методология в главата "Как да решаваме задачи по програмиране".

Задача 1: Извличане на текста от HTML документ

Даден е HTML файл с име **`Problem1.html`**. Да се напише програма, която отстранява от него всички HTML тагове и запазва само текста вътре в тях. Изходът да се изведе във файла **`Problem1.txt`**.

Примерен входен файл **`Problem1.html`**:

```
<html>
<head><title>Welcome to our site!</title></head>
<body>
<center>
<img src="/en/img/logo.gif" width="130" height="70" alt="Logo">
<br><br><br>
<font size="-1"><a href="/index.html">Home</a>
<a href="/contacts.html">Contacts</a>
<a href="/about.html">About</a></font><p>
</center>
</body>
</html>
```

Примерен изходен файл **`Problem1.txt`**:

```
Welcome to our site!
Home
Contacts
About
```

Измисляне на идея за решение

Първото, което ни хрумва като идея за решение на тази задача, е да четем последователно (примерно ред по ред) входния файл и да махаме всички тагове. Лесно се вижда, че всички тагове започват със символа "**<**" и завършват със символа "**>**". Това се отнася и за отварящите и за затварящите тагове. Това означава, че от всеки ред във файла трябва да се премахнат всички поднизове, започващи с "**<**" и завършващи с "**>**".

Проверка на идеята

Имаме идея за решаване на задачата. Дали идеята е вярна? Първо трябва да я проверим. Можем да я проверим дали е вярна за примерния входен файл, а след това да помислим дали няма някакви специални случаи, за които идеята би могла да е некоректна.

Взимаме лист и химикал и проверяваме на ръка идеята дали е вярна. Задраскваме всички поднизове от текста, които започват със символа "**<**"

и завършват със символа ">". Като го направим, виждаме, че остава само чистият текст и всички тагове изчезват:

```
~~<html>~~
~~<head><title>~~Welcome to our site!~~</title></head>~~
~~<body>~~
~~<center>~~
~~<img src="/en/img/logo.gif" width="130" height="70" alt="Logo">~~
~~<br><br><br>~~
~~<font size="-1"><a href="/index.html">~~Home~~</a>~~
~~<a href="/contacts.html">~~Contacts~~</a>~~
~~<a href="/about.html">~~About~~</a></font><p>~~
~~</center>~~
~~</body>~~
~~</html>~~
```

Сега остава да измислим някакви по-специални случаи. Нали не искаме да напишем 200 реда код и чак тогава да се сетим за тях и да трябва да преправяме цялата програма? Затова е важно да проверим проблемните ситуации, още сега, преди да сме почнали да пишем кода на решението.

Можем да се сетим за следния специален пример:

```
<html><body>
Click<a href="info.html">on this
link</a>for more info.<br />
This is<b>bold</b>text.
</body></html>
```

В него има две особености:

- Има тагове, съдържащи текст, които се отварят и затварят на различни редове. Такива тагове в нашия пример са <html>, <body> и <a>.
- Има тагове, които съдържат текст и други тагове в себе си. Например <body> и <html>.

Какъв трябва да е резултатът за този пример? Ако директно махнем всички тагове, ще получим нещо такова:

```
Clickon this
linkfor more info.
This isboldtext.
```

Или може би трябва да следваме правилата на езика HTML и да получим следния текст:

```
Click on this link for more info.
This is bold text.
```

Има и други варианти, например да слагаме всяко парче текст, което не е таг, на нов ред:

```
Click
on this
link
for more info.
This is
bold
text.
```

Ако махнем всичкия текст в таговете и долепим останалия текст, ще получим думи, които са залепени една до друга. От условието на задачата не става ясно дали това е исканият резултат или трябва, както в езика HTML, да получим по един интервал между отделните тагове. В езика HTML всяка поредица от разделители (интервали, нов ред, табулации и др.) се визуализира като един интервал. Това, обаче, не е споменато в условието на задачата и не става ясно от примерния вход и изход.

Не става ясно още дали трябва да отпечатваме думите, които са в таг, съдържащ в себе си други тагове или да ги пропускаме. Ако отпечатваме само съдържанието на тагове, в които има единствено текст, ще получим нещо такова:

```
on this
link
bold
```

От условието не става ясно още как се визуализира текст, който е разположен на няколко реда във вътрешността на някой таг.

Изясняване на условието на задачата

Първото, което трябва да направим, когато открием неясен момент в условието на задачата, е да го прочетем внимателно. В случая условието наистина не е ясно и не ни дава отговор на въпросите. Най-вероятно не трябва да следваме HTML правилата, защото те не са описани в условието, но не става ясно дали долепяме думите в съседни тагове или да ги разделяме с нов ред.

Остава ни само едно: да питаме. Ако сме на изпит, ще питаме този, който ни е дал задачите. Ако сме в реалния живот, то все някой е поръчител на софтуера, който разработваме, и той би могъл да отговори на възникналите въпроси. Ако никой не може да отговори, избираме един от вариантите, който ни се струва най-правилен съгласно условието на задачата и действаме по него.

Приемаме, че трябва да се отпечата текста, който остава като премахнем всички отварящи и затварящи тагове, като използваме за разделител празен ред. Ако в текста има празни редове, запазваме ги. За нашия пример трябва да получим следния изход:

```
Click
on this
link
for more info.
This is
bold
text.
```

Нова идея за решаване на задачата

И така, нашата адаптирана към новите изисквания идея е следната: четем файла ред по ред и във всеки ред заместваме таговете с нов ред. За да избегнем дублирането на нови редове в резултатния файл, заместваме всеки два последователни нови реда от резултата с един нов ред.

Проверяваме новата идея с оригиналния пример от условието на задачата и с нашия пример и се убеждаваме, че идеята този път е вярна. Остава да я реализираме.

Разбиваме задачата на подзадачи

Задачата лесно можем да разбием на подзадачи:

- Прочитане на входния файл.
- Обработка на един ред от входния файл: заместване на таговете със символ за нов ред.
- Записване на резултата в изходния файл.

Какво структури от данни да използваме?

В тази задача трябва да извършваме проста текстообработка и работа с файлове. Въпросът какви структури от данни да ползваме не стои пред нас – за текстообработката ще ползваме **`string`** и ако се наложи – **`StringBuilder`**.

Да помислим за ефективността

Ако четем редовете един по един, това няма да е бавна операция. Самата обработка на един ред може да се извърши чрез заместване на символи с други – също бърза операция. Не би трябвало да имаме проблеми с производителността.

Може би проблеми ще създаде изчистването на празните редове. Ако събираме всички редове в някакъв буфер (**`StringBuilder`**) и след това премахваме двойните празни редове, този буфер ще заеме много памет при големи входни файлове (примерно при 500 MB входен файл).

За да спестим памет, ще се опитаме да чистим излишните празни редове още след заместване на таговете със символа за празен ред.

Вече разгледахме внимателно идеята за решаване на задачата, уверихме се, че е добра и покрива специалните случаи, които могат да възникнат, и смятаме, че няма да имаме проблеми с производителността. Сега вече можем спокойно да преминем към имплементация на алгоритъма. Ще пишем кода стъпка по стъпка, за да откриваме грешките възможно най-рано.

Стъпка 1 – прочитане на входния файл

Първата стъпка от решението на поставената задача е прочитането входния файл. В нашия случай той е HTML файл. Това не трябва да ни притеснява, тъй като HTML е текстов формат. Затова, за да го прочетем, ще използваме класа **`StreamReader`**. Ще обходим входния файл ред по ред и за всеки ред ще извличаме (засега не ни интересува как) нужната ни информация (ако има) и ще я записваме в обект от тип **`StringBuilder`**. Извличането ще реализираме в следващата стъпка (стъпка 2), а записването в някоя от по-следващите стъпки. Да напишем нужния код за реализацията на нашата първа стъпка:

```
string line = string.Empty;
StreamReader reader = new StreamReader("Problem1.html");

while ((line = reader.ReadLine()) != null)
{
   // Find what we need and save it in the result
}

reader.Close();
```

Чрез написания код ще прочетем входния файл ред по ред. Да помислим дали сме реализирали добре първата стъпка. Сещате ли се какво пропуснахме?

С написаното ще прочетем входния файл, но само ако съществува. Ами ако входния файл не съществува или не може да бъде отворен по някаква причина? Сегашното ни решение няма да се справи с тези проблеми. В кода има и още един проблем: ако настъпи грешка при четенето или обработката на данните от файла, той няма да бъде затворен.

Чрез **`File.Exists(…)`** ще проверяваме дали входния файл съществува. Ако не съществува ще извеждаме подходящо съобщение и ще прекратяваме изпълнението на програмата. За да избегнем втория проблем ще използваме конструкцията **`try-catch-finally`**. Така, ако възникне изключение ще го обработим и накрая винаги ще затваряме файла, с които сме работили. Не трябва да забравяме, че обекта от **`StreamReader`** трябва да е деклариран извън **`try`** блока, защото иначе ще е недостъпен във **`finally`** блока. Това не е фатална грешка, но често се допуска от начинаещите програмисти.

Добре е да дефинираме името на входния файл като константа, защото вероятно ще го използваме на няколко места.

Още нещо: при четене от текстов файл е редно да зададем кодирането на файла. В случая ще използваме кодиране **`windows-1251`**, тъй като искаме да поддържаме коректно работа с български уеб сайтове на кирилица.

Да видим до какво стигнахме:

```
using System.IO;
using System.Text;

class HtmlTagRemover
{
    private const string InputFileName = "Problem1.html";
    private const string Charset = "windows-1251";

    static void Main()
    {
        StreamReader reader = null;
        Encoding encoding = Encoding.GetEncoding(Charset);
        string line = string.Empty;
        StringBuilder result = new StringBuilder();

        if (!File.Exists(InputFileName))
        {
            Console.WriteLine(
                "File " + InputFileName + " not found.");
            return;
        }

        try
        {
            reader = new StreamReader(InputFileName, encoding);
            while ((line = reader.ReadLine()) != null)
            {
                // Find what we need and save it in the result
            }
        }
        catch (IOException ioex)
        {
            Console.WriteLine(
                "Can not read file " + InputFileName + ".");
        }
```

```
      finally
      {
        if (reader != null)
        {
          reader.Close();
        }
      }
    }
  }
```

Справихме се с описаните проблеми и изглежда вече имаме коректно реализирано четенето на входния файл. За да сме напълно сигурни можем да тестваме. Например да изпишем съдържанието на входния файл на конзолата, а след това да пробваме с несъществуващ файл. Изписването ще става в **`while`** цикъла чрез **`Console.WriteLine(…)`**.

Ако тестваме с примера от условието на задачата, резултатът е следният:

```
<html>
<head>
<title>Welcome to our site!</title>
</head>
<body>
<center>
<img src="/en/img/logo.gif" width="130" height="70" alt="Logo">
<br><br><br>
<font size="-1"><a href="/index.html">Home</a> -
<a href="/contenst.html">Contacts</a> -
<a href="/about.html">About</a></font><p>
</center>
</body>
</html>
```

Да пробваме с несъществуващ файл. Да заменим името на файла **`Problem1.html`** с **`Problem2.html`**. Резултатът от това е следният:

```
File Problem2.html not found
```

Уверихме се, че дотук написаният код е верен. Да преминем към следващата стъпка.

Стъпка 2 – премахване на таговете

Сега трябва да измислим подходящ начин да премахнем всички тагове. Какъв да бъде начинът?

Един възможен начин е като проверяваме реда символ по символ. За всеки символ от текущия ред ще търсим символа "<". От него надясно ще знаем, че е имаме някакъв таг (отварящ или затварящ). Краят на тага символът ">". Така можем да откриваме таговете и да ги премахваме. За да не получим долепяне на думите в съседни тагове, ще заместваме всеки таг със символа за празен ред "**\n**".

Алгоритъмът не е сложен за имплементиране, но дали няма по-хитър начин? Можем ли да използваме регулярни изрази? С тях лесно можем да търсим тагове и да ги заместваме с "**\n**", нали? Същевременно кодът няма да е сложен и при възникване на грешки по-лесно ще бъдат отстранени. Ще се спрем на този вариант. Какво трябва да направим? Първо трябва да напишем регулярния израз. Ето как изглежда той:

```
<[^>]*>
```

Идеята е проста: всеки низ, който започва с "**<**", продължава с произволи символи, различни от "**>**" и завършва с "**>**", е HTML таг. Ето как можем да заместим таговете със символ за нов ред:

```
private static string RemoveAllTags(string str)
{
   string textWithoutTags = Regex.Replace(str, "<[^>]*>", "\n");
   return textWithoutTags;
}
```

След като написахме тази стъпка, трябва да я тестваме. За целта отново ще изписваме намерените низове на конзолата чрез **`Console.WriteLine(…)`**. Да тестваме кода, който получихме:

HtmlTagRemover.cs

```
using System.IO;
using System.Text.RegularExpressions;

class HtmlTagRemover
{
   private const string InputFileName = "Problem1.html";
   private const string Charset = "windows-1251";

   static void Main()
   {
      StreamReader reader = null;
      Encoding encoding = Encoding.GetEncoding(Charset);
      string line = string.Empty;
      StringBuilder result = new StringBuilder();

      if (!File.Exists(InputFileName))
      {
         Console.WriteLine(
            "File " + InputFileName + " not found.");
```

```
        return;
      }

      try
      {
        reader = new StreamReader(InputFileName, encoding);
        while ((line = reader.ReadLine()) != null)
        {
          line = RemoveAllTags(line);
          Console.WriteLine(line);
        }
      }
      catch (IOException ioex)
      {
        Console.WriteLine(
          "Can not read file " + InputFileName + ".");
      }
      finally
      {
        if (reader != null)
        {
          reader.Close();
        }
      }
    }

    private static string RemoveAllTags(string str)
    {
      string strWithoutTags = Regex.Replace(str, "<[^>]*>",
        "\n");
      return strWithoutTags;
    }
}
```

Да стартираме програмата със следния входен файл:

```
<html><body>
Click<a href="info.html">on this
link</a>for more info.<br />
This is<b>bold</b>text.
</body></html>
```

Резултатът ще бъде е следният:

```
(празени редове)
Click
on this
link
for more info.
```

```
(празен ред)
This is
bold
text.
(празни редове)
```

Всичко работи отлично, само че имаме излишни празни редове. Можем ли да ги премахнем? Това ще е следващата ни стъпка.

Стъпка 3 – премахване на празните редове

Можем да премахнем излишните празни редове, като заменяме двоен празен ред "**\n\n**" с единичен празен ред "**\n**". Два символа за нов ред един след друг означават преминаване на нов ред и още едно преминаване на нов ред, при което се получава празен ред. Затова не трябва да имаме такива струпвания на повече от един символ за нов ред **\n**. Ето примерен метод, който извършва замяната:

```
private static string RemoveDoubleNewLines(string str)
{
   return str.Replace("\n\n", "\n");
}
```

Както, винаги, преди да продължим напред, тестваме метода дали работи коректно. Пробваме с текст, в който няма празни редове, а след това добавяме 2, 3, 4 и 5 празни реда, включително в началото и в края на текста.

Установяваме, че методът не работи коректно, когато има 4 празни реда един след друг. Например ако подадем като входни данни "**ab\n\n\n\ncd**", получаваме "**ab\n\n\cd**" вместо "**ab\ncd**". Този дефект се получава, защото **Replace(…)** намира и замества съвпаденията еднократно отляво надясно. Ако в резултат на заместване се появи отново търсеният низ, той бива прескочен.

Видяхте колко е полезно всеки метод да бъде тестван на момента, а не накрая да се чудим защо програмата не работи и да имаме 200 реда код, пълен с грешки и да се чудим от къде идва проблема. Ранното откриване на дефектите е много полезно и трябва да го правите винаги, когато е възможно. Ето поправения код:

```
private static string RemoveDoubleNewLines(string str)
{
   string pattern = "[\n]+";
   return Regex.Replace(str, pattern, "\n");
}
```

След серия тестове, се убеждаваме, че сега вече методът работи коректно. Готови сме да тестваме цялата програма дали отстранява коректно излишните нови редове. За целта правим следната промяна:

```
while ((line = reader.ReadLine()) != null)
{
   line = RemoveAllTags(line);
   line = RemoveDoubleNewLines(line);
   Console.WriteLine(line);
}
```

Изглежда пак има празни редове. От къде ли идват? Вероятно, ако имаме ред, който съдържа само тагове, той ще създаде проблем. Следователно трябва да предвидим този случай. Добавяме следната проверка:

```
if (!string.IsNullOrEmpty(line))
{
   Console.WriteLine(line);
}
```

Това премахва повечето празни редове, но не всички.

Ако се замислим, би могло да се случи така, че някой ред да започва или завършва с таг. Тогава този таг ще бъде заменен с единичен празен ред и така в началото или в края на реда може да има празен ред. Това означава, че трябва да чистим празните редове в началото и в края на всеки ред. Ето как можем да направим въпросното изчистване:

```
private static string TrimNewLines(string str)
{
   int start = 0;
   while (start < str.Length && str[start] == '\n')
   {
      start++;
   }

   int end = str.Length - 1;
   while (end >= 0 && str[end] == '\n')
   {
      end--;
   }

   if (start > end)
   {
      return string.Empty;
   }

   string trimmed = str.Substring(start, end - start + 1);
   return trimmed;
}
```

Методът работи много просто: преминава отляво надясно пред входния символен низ и прескача всички символи за празен ред. След това преминава отдясно наляво и отново прескача всички символи за празен ред. Ако лявата и дясната позиция са се разминали, това означава, че низът или е празен, или съдържа само символи за празен ред. Тогава връщаме празен низ. Иначе връщаме всичко надясно от стартовата позиция и наляво от крайната позиция.

Както винаги, тестваме въпросния метод дали работи коректно с няколко примера, сред които празен низ, низ без нови редове, низ с нови редове отляво или отдясно или и от двете страни и низ само с нови редове. Убеждаваме се, че методът работи коректно.

Сега остава да модифицираме логиката на обработката на входния файл:

```
while ((line = reader.ReadLine()) != null)
{
   line = RemoveAllTags(line);
   line = RemoveDoubleNewLines(line);
   line = TrimNewLines(line);
   if (!string.IsNullOrEmpty(line))
   {
      Console.WriteLine(line);
   }
}
```

Този път тестваме и се убеждаваме, че всичко работи коректно.

Стъпка 4 – записване на резултата във файл

Остава ни да запишем резултата в изходен файл. За да записваме резултата в изходния файл ще използваме **`StreamWriter`**. Тази стъпка е тривиална. Трябва да се съобразим само, че писането във файл може да предизвика изключение и затова трябва да променим леко логиката за обработка на грешки и за отварянето и затварянето на потоците за входния и изходния файл.

Ето какво се получава най-накрая като изходен код на програмата:

HtmlTagRemover.cs

```
using System.IO;
using System.Text.RegularExpressions;

class HtmlTagRemover
{
   private const string InputFileName = "Problem1.html";
private const string OutputFileName = "Problem1.txt";
private const string Charset = "windows-1251";

static void Main()
{
  StreamReader reader = null;
  StreamWriter writer = null;
  Encoding encoding = Encoding.GetEncoding(Charset);
  string line = string.Empty;
  StringBuilder result = new StringBuilder();

  if (!File.Exists(InputFileName))
  {
    Console.WriteLine(
      "File " + InputFileName + " not found.");
    return;
  }

  try
  {
    reader = new StreamReader(InputFileName, encoding);
    writer = new StreamWriter(OutputFileName, false,
      encoding);

    while ((line = reader.ReadLine()) != null)
    {
      line = RemoveAllTags(line);
      line = RemoveDoubleNewLines(line);
      line = TrimNewLines(line);
      if (!string.IsNullOrEmpty(line))
      {
        writer.WriteLine(line);
      }

    }
  }
  catch (IOException ioex)
  {
    Console.WriteLine(
      "Can not read file " + InputFileName + ".");
  }
  finally
  {
    if (reader != null)
    {
      reader.Close();
    }

    if (writer != null)
```

```
        {
            writer.Close();
        }
    }
}

/// <summary>
/// Replaces every tag with new line
/// </summary>
private static string RemoveAllTags(string str)
{
    string strWithoutTags = Regex.Replace(str, "<[^>]*>",
        "\n");
    return strWithoutTags;
}

/// <summary>
/// Replaces sequence of new lines with only one new line
/// </summary>
private static string RemoveDoubleNewLines(string str)
{
    string pattern = "[\n]+";
    return Regex.Replace(str, pattern, "\n");
}

/// <summary>
/// Removes new lines from start and end of string
/// </summary>
private static string TrimNewLines(string str)
{
    int start = 0;
    while (start < str.Length && str[start] == '\n')
    {
        start++;
    }

    int end = str.Length - 1;
    while (end >= 0 && str[end] == '\n')
    {
        end--;
    }

    if (start > end)
    {
        return string.Empty;
    }

    string trimmed = str.Substring(start, end - start + 1);
    return trimmed;
```

```
  }
}
```

Тестване на решението

Досега тествахме отделните стъпки от решението на задачата. Чрез извършените тестове на отделните стъпки намаляваме възможността за грешки, но това не значи, че не трябва да тестваме цялото решение. Може да сме пропуснали нещо, нали?

Тестваме с примерния входен файл от условието на задачата. Всичко работи коректно.

Тестваме с нашия "сложен" пример. Всичко работи добре.

Задължително трябва да тестваме граничните случаи и да пуснем тест за производителност.

Започваме с празен файл. Изходът е коректен – празен файл.

Тестваме с файл, който съдържа само една дума "**`Hello`**" и не съдържа тагове. Резултатът е коректен – изходът съдържа само думата "**`Hello`**".

Тестваме с файл, който съдържа само тагове и не съдържа текст. Резултатът е отново коректен – празен файл.

Пробваме да сложим празни редове на най-невероятни места във входния файл. Пускаме следния тест:

```
  Hello

<br><br>


<b>I<b> am here

I am not <b>here</b>
```

Изходът е следният:

```
  Hello
I
 am here
I am not
here
```

Изглежда открихме дребен дефект. Има един интервал в началото на два от редовете. Според условието не е много ясно дали това е дефект, но нека се опитаме да го оправим.

Добавяме следния код при обработката на поредния ред от входния файл:

```
line = line.Trim();
```

Дефектът се премахва, но само на първия ред. Пускаме дебъгера и забелязваме защо се получава така. Причината е, че отпечатваме в изходния файл символен низ със стойност "**`I\n am here`**" и така получаваме интервал след празен ред. Можем да поправим дефекта, като навсякъде заместим празен ред, следван от празно пространство (празни редове, интервали, табулации и т.н.), с единичен празен ред. Ето как изглежда поправката:

```
private static string RemoveDoubleNewLines(string str)
{
   string pattern = "\n\\s+";
   return Regex.Replace(str, pattern, "\n");
}
```

Поправихме и тази грешка. Единствено трябва да му сменим името с някакво по-адекватно като например **`RemoveNewLinesWithWhiteSpace()`**.

Сега трябва отново да тестваме упорито след поправката. Слагаме нови редове и интервали пръснати безразборно и се уверяваме се, че всичко работи вече коректно.

Остана един последен тест – за производителност. Лесно можем да създадем обемен входен файл. Отваряме някой сайт, примерно http://www.microsoft.com, взимаме сорс кода му и го копираме 1000 пъти. Получаваме достатъчно голям входен файл. В нашия случай се получи 44 MB файл с 947 000 реда. За обработката му бяха нужни под 10 секунди, което е напълно приемлива скорост. Когато тестваме решението не трябва да забравяме, че обработката на файла зависи от компютърната ни конфигурация.

Като надникнем в резултата, обаче, забелязваме много неприятен проблем. В него има части от тагове. По-точно виждаме следното:

```
<!--
var s_pageName="home page"
//-->
```

Бързо става ясно, че сме изпуснали един много интересен случай. В HTML може един таг да бъде затворен няколко реда след отварянето си, т.е. един таг може да е разположен на няколко последователни реда. Точно такъв е нашият случай: имаме таг с коментари, който съдържа JavaScript код. Ако програмата работеше коректно, щеше да отреже целия таг вместо да го запази в изходния файл.

Видяхте колко е полезно тестването и колко е важно. В някои сериозни фирми (като например Майкрософт) решение без тестове се счита за готово на 50%. Това означава, че ако пишете код 2 часа, трябва да отделите за тестване (ръчно или автоматизирано) поне още 2 часа! Само така можете да създадете качествен софтуер.

Колко жалко, че открихме проблема чак сега вместо в началото, когато проверявахме дали е правилна идеята ни за решение на задачата, преди да сме написали програмата. Понякога се случва така, няма как.

Как да оправим проблема с тагове на два реда?

Първата идея, която ни хрумва, е да заредим в паметта целия входен файл и да го обработваме като един голям стринг вместо ред по ред. Това е идея, която изглежда ще работи, но ще работи бавно и ще консумира голямо количество памет. Нека потърсим друга идея.

Очевидно не можем да четем файла ред по ред. Можем ли да го четем символ по символ? Ако можем, как ще обработваме таговете? Хрумва ни, че ако четем файла символ по символ, можем във всеки един момент да знаем дали сме в таг или сме извън таг и ако сме извън таг, можем да печатаме всичко, което прочетем. Ще се получи нещо такова:

```
bool inTag = false;
while (! <end of file is reached>)
{
   char ch = <read next character>;
   if (ch == '<')
   {
      inTag = true;
   }
   else if (ch == '>')
   {
      inTag = false;
   }
   else
   {
      if (!inTag)
      {
         PrintBuffer(ch);
      }
   }
}
```

Идеята е много проста и лесна за реализация. Ако я реализираме директно, ще имаме проблема с празните редове и проблема със сливането на текст от съседни тагове. За да разрешим този проблем, можем да натрупваме текста в **StringBuilder** и да го отпечатваме при край на файла или при преминаване към таг. Ще се получи нещо такова:

```
bool inTag = false;
StringBuilder buffer = new StringBuilder();
```

```
while (! <end of file is reached>)
{
   char ch = <read next character>;
   if (ch == '<')
   {
      if (!inTag)
      {
         PrintBuffer(buffer);
      }
      buffer.Remove(0, buffer.Length);
      inTag = true;
   }
   else if (ch == '>')
   {
      inTag = false;
   }
   else
   {
      if (!inTag)
      {
         buffer.Append(ch);
      }
   }
}
PrintBuffer(buffer);
```

Ако добавим и логиката за избягване на празните редове, както и четенето на входа и писането на резултата, ще получим цялостно решение на задачата по новия алгоритъм:

```
using System.IO;
using System.Text.RegularExpressions;

public class SimpleHtmlTagRemover
{
   private const string InputFileName = "Problem1.html";
   private const string OutputFileName = "Problem1.txt";
   private const string Charset = "windows-1251";

   public static void Main()
   {
      StringBuilder buffer = new StringBuilder();
      bool inTag = false;

      if (!File.Exists(InputFileName))
      {
         Console.WriteLine(
            "File " + InputFileName + " not found.");
         return;
```

```
    }

    StreamReader reader = null;
    StreamWriter writer = null;
    try
    {
      Encoding encoding = Encoding.GetEncoding(Charset);
      reader = new StreamReader(InputFileName, encoding);
      writer = new StreamWriter(OutputFileName, false,
        encoding);

      Regex regex = new Regex("\n\\s+");
      while (true)
      {
        int nextChar = reader.Read();
        if (nextChar == -1)
        {
          // End of file reached
          PrintBuffer(writer, buffer, regex);
          break;
        }
        char ch = (char)nextChar;
        if (ch == '<')
        {
          if (!inTag)
          {
            PrintBuffer(writer, buffer, regex);
          }
          buffer.Length = 0;
          inTag = true;
        }
        else if (ch == '>')
        {
          inTag = false;
        }
        else
        {
          // We have other character
          // (not "<" or ">")
          if (!inTag)
          {
            buffer.Append(ch);
          }
        }

      }
    }
    catch (IOException ioex)
    {
            Console.WriteLine(
               "Can not read file " + InputFileName + ".");
         }
         finally
         {
            if (reader != null)
            {
               reader.Close();
            }

            if (writer != null)
            {
               writer.Close();
            }
         }
      }

      /// <summary>
      /// Remove intervals and prints the buffer in file
      /// </summary>
      /// <param name="writer">store the result in file</param>
      /// <param name="buffer">keeps the current result from
      /// html file</param>
      /// <param name="regex">removes new lines followed by
      /// whitespaces</param>
      private static void PrintBuffer(StreamWriter writer,
            StringBuilder buffer, Regex regex)
      {
         string str = buffer.ToString();
         string trimmed = str.Trim();
         string textOnly = regex.Replace(trimmed, "\n");
         if (!string.IsNullOrEmpty(textOnly))
         {
            writer.WriteLine(textOnly);
         }
      }
   }
```

Входният файл чете символ по символ с класа **`StreamReader`**.

Първоначално буферът за натрупване на текст е празен. В главния цикъл анализираме всеки прочетен символ. Имаме следните случаи:

- Ако стигнем до края на файла, отпечатваме каквото има в буфера и алгоритъмът приключва.
- При срещане на символ "**<**" (начало на таг) първо отпечатваме буфера (ако установим, че преминаваме от текст към таг). След това зачистваме буфера и установяваме **`inTag = true`**.

- При срещане на символ ">" (край на таг) установяваме **`inTag = false`**. Това ще позволи следващите след тага символи да се натрупват в буфера.
- При срещане на някой друг символ (текст или празно пространство), той се добавя към буфера, ако сме извън таг. Ако сме в таг, символът се игнорира.

Печатането на буфера се грижи да премахва празните редове в текста и да изчиства празното пространство в началото и в края на текста. Как точно извършваме това, вече разгледахме в предходното решение на задачата, което се оказа грешно.

Във второто решение обработката на буфера е много по-лека и по-кратка, затова буфера се обработва непосредствено преди отпечатването му.

В предишното решение на задачата използваме регулярни изрази за заместване чрез статичните методи на класа **`Regex`**, а сега създаваме обект от същия клас. При статичния вариант шаблонът се компилира за машината на регулярните изрази всеки път, иначе само веднъж и просто се подава като аргумент на нужните методи.

Тестване на новото решение

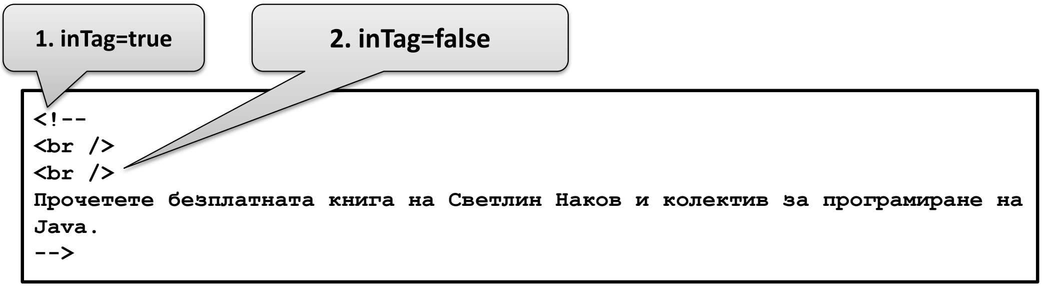
Остава да тестваме задълбочено новото решение. Изпълняваме всички тестове, които проведохме за предното решение. Добавяме тест с тагове, които се разпростират на няколко реда. Отново тестваме за производителност със сайта на Майкрософт 1000 пъти. Уверяваме се, че и за него програмата работи коректно и дори е по-бърза.

Нека да пробваме с някой друг сайт, например официалният Интернет сайт на книгите по въведение в програмирането със C# и Java: http://www.introprogramming.info/. Отново взимаме сорс кода му и пускаме решението на нашата задача. След внимателно преглеждане на входните данни (сорс кода на сайта на книгата) и изходния файл, забелязваме че отново има проблем. Част от съдържанието от този таг се отпечатва в изходния файл:

```
<!--
<br />
<br />
Прочетете безплатната книга на Светлин Наков и колектив за
програмиране на Java.
...
...
-->
```

Къде е проблемът?

Проблемът изглежда възниква, когато в един таг се среща друг таг преди първият да бъде затворен. Това може да се случи при HTML коментарите. Ето как се стига до грешката:



```
<!--
<br />
<br />
Прочетете безплатната книга на Светлин Наков и колектив за програмиране на
Java.
-->
```

Както знаем в решението на задачата използваме булева променлива (**`inTag`**), за да знаем дали текущия символ се намира в таг или не. На картинката сме показали, че в момент 1 установяваме **`inTag = true`**. Дотук изпълнението на задачата е нормално. Настъпва в момент 2, където текущия прочетен символ е "**>**". В този момент установяваме **`inTag = false`**. Проблема е, че тагът, който е отворен от момент 1 все още не е затворен, а булевата променлива показва, че вече не сме в таг и следващите символи се записват в буфера. Ако между двата тага за нов ред (<br />) имаше текст, той също щеше да бъде записан в буфера.

Как да оправим проблема?

Оказа се, че и във второто решение има грешка. Програмата не работи коректно при наличието на вложени тагове в таг с коментари. Чрез булева променлива може да знаем само дали сме в таг или не, но не и да помним дали все още дали сме в предходния. Това ни подсказва, че вместо да използваме булева променлива може да запомняме броя на таговете, в които се намираме (променлива от тип **`int`**). Ще модифицираме предходното решение:

```
int openedTags = 0;
StringBuilder buffer = new StringBuilder();
while (! <end of file is reached>)
{
   char ch = <read next character>;
   if (ch == '<')
   {
      if (openedTags == 0)
      {
         PrintBuffer(buffer);
      }
      buffer.Remove(0, buffer.Length);
```

```
        openedTags++;
    }
    else if (ch == '>')
    {
        openedTags--;
    }
    else
    {
        if (openedTags == 0)
        {
            buffer.Append(ch);
        }
    }
}
PrintBuffer(buffer);
```

В главния цикъл анализираме всеки прочетен символ. Имаме следните случаи:

- Ако стигнем до края на файла, отпечатваме каквото има в буфера и алгоритъмът приключва.
- При срещане на символ "<" (начало на таг) първо отпечатваме буфера (ако установим, че преминаваме от текст към таг). След това зачистваме буфера и увеличаваме брояча с единица.
- При срещане на символ ">" (край на таг) намаляваме брояча с единица. Затварянето на вложен таг няма да позволи натрупване в буфера. Ако след затварянето на таг сме извън тагове символите ще започнат да се натрупват в буфера.
- При срещане на някой друг символ (текст или празно пространство), той се добавя към буфера, ако сме извън таг. Ако сме в таг, символът се игнорира.

Остава ни да напишем цялото решение и след това да тестваме. Логиката по четенето на входния файл и печатането на буфера остава същата:

```
using System.IO;
using System.Text.RegularExpressions;

public class SimpleHtmlTagRemover
{
    private const string InputFileName = "Problem1.html";
    private const string OutputFileName = "Problem1.txt";
    private const string Charset = "windows-1251";

    public static void Main()
    {
        StringBuilder buffer = new StringBuilder();
    int openedTags = 0;

    if (!File.Exists(InputFileName))
    {
        Console.WriteLine(
            "File " + InputFileName + " not found.");
        return;
    }

    StreamReader reader = null;
    StreamWriter writer = null;
    try
    {
        Encoding encoding = Encoding.GetEncoding(Charset);
        reader = new StreamReader(InputFileName, encoding);
        writer = new StreamWriter(OutputFileName, false,
            encoding);

        Regex regex = new Regex("\n\\s+");
        while (true)
        {
            int nextChar = reader.Read();
            if (nextChar == -1)
            {
                // End of file reached
                PrintBuffer(writer, buffer, regex);
                break;
            }
            char ch = (char)nextChar;
            if (ch == '<')
            {
                if (openedTags == 0)
                {
                    PrintBuffer(writer, buffer, regex);
                    buffer.Length = 0;
                }
                openedTags++;
            }
            else if (ch == '>')
            {
                openedTags--;

            }
            else
            {
                // We aren't in tags (not "<" or ">")
                if (openedTags == 0)
                {
                    buffer.Append(ch);
                }
            }
        }
    }
    catch (IOException ioex)
    {
        Console.WriteLine(
            "Can not read file " + InputFileName + ".");
    }
    finally
    {
        if (reader != null)
        {
            reader.Close();
        }

        if (writer != null)
        {
            writer.Close();
        }
    }
}

/// <summary>
/// Remove intervals and prints the buffer in file
/// </summary>
/// <param name="writer">store the result in file</param>
/// <param name="buffer">keeps the current result from
/// html file</param>
/// <param name="regex">removes new lines followed by
/// whitespaces</param>
private static void PrintBuffer(StreamWriter writer,
        StringBuilder buffer, Regex regex)
{
    string str = buffer.ToString();
    string trimmed = str.Trim();
    string textOnly = regex.Replace(trimmed, "\n");
    if (!string.IsNullOrEmpty(textOnly))
    {
        writer.WriteLine(textOnly);
    }
}
}
```

Тестване на новото решение

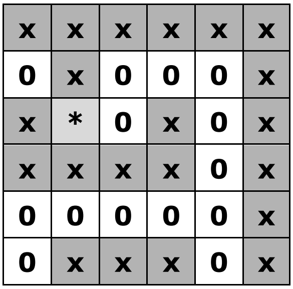
Отново тестваме решението на задачата. Изпълняваме всички тестове, направени за предното решение (вж. секцията "Тестване на решението"). Да пробваме със сайта на MSDN (http://msdn.microsoft.com/). Нека да проверим внимателно изходния файл. Може да се види, че в края на файла има грешни символи. След като внимателно прегледаме сорс кода на сайта на MSDN забелязваме, че има неправилно изобразяване на символа ">" (за да се визуализира този символ в HTML документ, трябва да се използва "**`&gt;`**", а не ">"). Грешката е на сайта на MSDN, а не в нашата програма.

Сега ни остава да тестваме за производителност със сайта на книгата (http://www.introprogramming.info), копиран 1000 пъти. Уверяваме се, че и за него програмата работи достатъчно бързо.

Най-сетне вече сме готови за следващата задача.

Задача 2: Лабиринт

Даден е лабиринт, който се състои от N x N квадратчета, всяко от които може да е проходимо (`0`) или не (**`x`**). В едно от квадратчетата се намира нашият герой Минчо (**`*`**):



x	x	x	x	x	x
0	x	0	0	0	x
x	*	0	x	0	x
x	x	x	x	0	x
0	0	0	0	0	x
0	x	x	x	0	x

Две квадратчета са съседни, ако имат обща стена. Минчо може на една стъпка да преминава от едно проходимо квадратче в съседно на него проходимо квадратче. Ако Минчо стъпи в клетка, която е на границата на лабиринта, той може с една стъпка да излезе извън него. Напишете програма, която по даден лабиринт отпечатва минималния брой стъпки, необходими на Минчо, за да излезе от лабиринта или -1 ако няма изход.

Входните данни се четат от текстов файл с име **`Problem2.in`**. На първия ред във файла стои числото N (2 < N < 100). На следващите N реда стоят по N символа, всеки от които е или "`0`" или "**`x`**" или "**`*`**". Изходът представлява едно число и трябва да се изведе във файла **`Problem2.out`**.

Примерен входен файл **`Problem2.in`**:

```
6
xxxxxx
0x000x
x*0x0x
xxxx0x
00000x
0xxx0x
```

Примерен изходен файл **`Problem2.out`**:

```
9
```

Измисляне на идея за решение

Имаме лабиринт и трябва да намерим най-краткия път в него. Това не е лесна задача и трябва доста да помислим или да сме прочели някъде как се решават такива задачи.

Нашият алгоритъм ще започва от работата си от началната точка, която ни е дадена. Знаем, че можем да се предвижваме в съседна клетка хоризонтално и вертикално, но не и по диагонал. Нашият алгоритъм трябва да обхожда лабиринта по някакъв начин, за да намери в него най-късия път. Как да обхождаме клетките в лабиринта?

Един възможен начин за обхождане е следният: стартираме от началната клетка. Преместваме се в съседна клетка на текущата (която е проходима), след това в съседна клетка на нея (която е проходима и все още непосетена), след това в съседна на последната посетена (която е проходима е все още непосетена) и така продължаваме рекурсивно напред, докато или стигнем изход от лабиринта, или стигнем до място, от където няма продължение (няма съседна клетка, която е свободна и непосетена). В този момент се връщаме от рекурсията (към предходната клетка, от която сме стигнали текущата) и посещаваме друга клетка на предходната клетка. Ако няма продължение, се връщаме още назад. Описаният рекурсивен процес представлява обхождане на лабиринта в дълбочина (спомнете си главата "Рекурсия").

Възниква въпросът "Нужно ли е да минаваме през една клетка повече от един път"? Ако минаваме през една клетка най-много веднъж, то бързо ще обходим целия лабиринт и ако има изход, ще го намерим. Обаче минимален ли ще е този път. Ако си нарисуваме процеса на хартия, бързо ще се убедим, че намереният път няма да е минимален.

Ако при връщане от рекурсията отбелязваме като свободна клетката, която напускаме, ще позволим до една и съща клетка да стигаме многократно, идвайки по различен път. Пълното рекурсивно обхождане на лабиринта на практика ще намери всички възможни пътища от началната клетка до всяка други клетка. От всички тези пътища можем да вземем най-късия път до клетка на границата на лабиринта (изход) и така ще намерим решение на задачата.

Проверка на идеята

Изглежда имаме идея за решаване на задачата: с рекурсивно обхождане намираме всички пътища в лабиринта от началната клетка до клетка на границата на лабиринта и измежду всички тези пътища избираме най-късия. Нека да проверим идеята.

Взимаме лист хартия и си правим един примерен лабиринт. Пробваме алгоритъма. Вижда се, че той намира всички пътища от началната клетка до някой от изходите, като доста обикаля напред-назад. В крайна сметка намира всички изходи и измежду всички пътища може да се избере най-краткият.

Дали идеята работи, ако няма изход? Правим си втори лабиринт, който е без изход. Пробваме алгоритъма върху него, отново на лист хартия. Виждаме, че след доста обикаляне напред-назад алгоритъмът не намира нито един изход и приключва.

Изглежда имаме правилна идея за решаване на задачата. Да преминем напред и да помислим за структурите от данни.

Какви структури от данни да използваме?

Първо трябва да преценим как да съхраняваме лабиринта. Съвсем естествено е да ползваме матрица от символи, точно като тази на картинката. Ще считаме, че една клетка е проходима и можем да влезем в нея, ако съдържа символ, различен от символа '**`x`**'. Може да пазим лабиринта и в матрица с числа или булеви стойности, но разликата не е съществена. Матрицата от символи е удобна за отпечатване, а това ще ни помогне докато дебъгваме. Няма много възможности за избор. Ще съхраняваме лабиринта в матрица от символи.

След това трябва да решим в каква структура да запомняме обходените до момента клетки по време на рекурсията (текущия път). На нас ни трябва винаги последната обходена клетка. Това ни навежда на мисълта за структура, която спазва "последен влязъл, пръв излязъл", тоест стек. Можем да ползваме **`Stack<Cell>`**, където **`Cell`** е клас, съдържащ координатите на една клетка (номер на ред и номер на колона).

Остава да помислим в какво да запомняме намерените пътища, за да можем да извлечем накрая най-късия от тях. Ако се замислим малко, не е нужно да пазим всички пътища. Достатъчно е да помним текущия път и най-късият път за момента. Дори не е необходимо да пазим най-късия път за момента, ами само неговата дължина. Всеки път, когато намерим път до изход от лабиринта, можем да взимаме неговата дължина и ако тя е по-малка от най-късата дължина за момента, да я запомняме.

Изглежда намерихме ефективни структури от данни. Според препоръките за решаване на задачи, още не трябва да се втурваме да пишем кода на програмата, защото трябва да помислим за ефективността на алгоритъма.

Да помислим за ефективността

Нека да проверим идеята си от следна точка на ефективността? Какво правим ние? Намираме всички възможни пътища и от тях взимаме най-късия. Няма спор, че алгоритъмът ще работи, но ако лабиринтът стане много голям, дали ще работи бързо?

За да отговорим на този въпрос, трябва да помислим колко са пътищата. Ако вземем празен лабиринт, то на всяка стъпка на рекурсията ще имаме средно по 3 свободни продължения (като изключим клетката, от която идваме).

Така, ако имаме примерно лабиринт 10 на 10, пътят може да стане дълъг до 100 клетки и по време на обхождането на всяка стъпка ще имаме по 3 съседни клетки. Изглежда броят пътища е число от порядъка на 3 на степен 100. Очевидно алгоритъмът ще "приспи" компютъра много бързо.

Намерихме сериозен проблем на алгоритъма. Той ще работи много бавно, дори при малки лабиринти, а при големи изобщо няма да работи! Хубавото е, че още не сме написали нито един ред код и генералната смяна на подхода към задачата няма да ни струва много пропиляно време.

Да измислим нова идея

Разбрахме, че обхождането на всички пътища в лабиринта е неправилен подход, затова трябва да измислим друг.

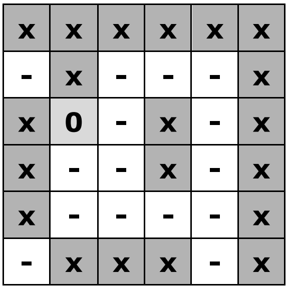
Нека започнем от началната клетка и да обходим всички нейни съседни и да ги маркираме като обходени. За всяка обходена клетка ще запомняме едно число, което е равно на броя клетки, през които сме преминали, за да достигнем до нея (дължина на пътя от началната клетка до текущата).

За началната клетка дължината на пътя е 0. За нейните съседи дължината на пътя трябва да е 1, защото с 1 движение можем да ги достигнем от началната клетка. За съседните клетки на съседите на началната клетка дължината на пътя е 2. Можем да продължим да разсъждаваме по този начин и ще стигнем до следния алгоритъм:

1. Записваме дължина на пътя 0 за началната клетка. Отбелязваме я като посетена.
2. За всяка съседна клетка на началната отбелязваме, че пътят до нея е с дължина 1. Отбелязваме тези клетки като посетени.
3. За всяка клетка, която е съседна на клетка с дължина 1 и не е посетена, записваме, че е дължината на пътя до нея е 2. Отбелязваме въпросните клетки като посетени.
4. Продължавайки аналогично, на N-тата стъпка намираме всички непосетени все още клетки, които са на разстояние N премествания от началната клетка и ги отбелязваме като посетени.

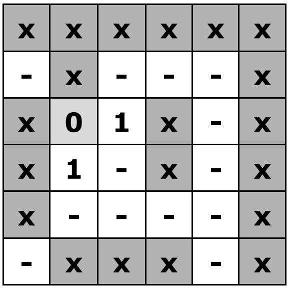
Можем да визуализираме процеса по следния начин (взимаме друг лабиринт, за да покажем по-добре идеята):

Стъпка 0 – отбелязваме разстоянието от началната клетка до нея самата с 0 (за удобство на картинката отбелязваме свободните клетки с "-"):



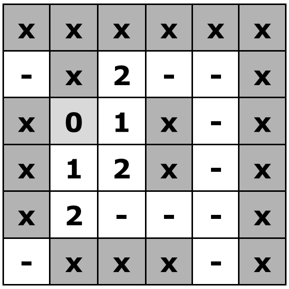
x	x	x	x	x	x
-	x	-	-	-	x
x	0	-	x	-	x
x	-	-	x	-	x
x	-	-	-	-	x
-	x	x	x	-	x

Стъпка 1 – отбелязваме с 1 всички проходими съседи на клетки със стойност 0:



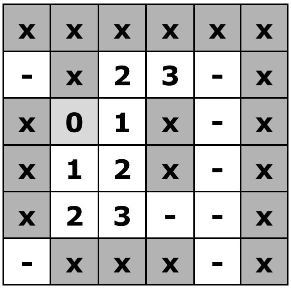
x	x	x	x	x	x
-	x	-	-	-	x
x	0	1	x	-	x
x	1	-	x	-	x
x	-	-	-	-	x
-	x	x	x	-	x

Стъпка 2 – отбелязваме с 2 всички проходими съседи на клетки с 1:



x	x	x	x	x	x
-	x	2	-	-	x
x	0	1	x	-	x
x	1	2	x	-	x
x	2	-	-	-	x
-	x	x	x	-	x

Стъпка 3 – отбелязваме с 3 всички проходими съседи на клетки със стойност 2:



x	x	x	x	x	x
-	x	2	3	-	x
x	0	1	x	-	x
x	1	2	x	-	x
x	2	3	-	-	x
-	x	x	x	-	x

Продължавайки така, в един момент или ще достигнем клетка на границата на лабиринта (т.е. изход) или ще установим, че такава не е достижима.

Проверяване производителността на новия алгоритъм

Понеже никога не посещаваме повече от веднъж една и съща клетка, броят стъпки, които извършва този алгоритъм, не би трябвало да е голям. Примерно, ако имаме лабиринт с размери 100 на 100, той ще има 10 000 клетки, всяка от които ще посетим най-много веднъж и за всяка посетена клетка ще проверим всеки неин съсед дали е свободен, т.е. ще извършим по 4 проверки. В крайна сметка ще извършим най-много 40 000 проверки и ще обходим най-много 10 000 клетки. Общо ще направим около 50 000 операции. Това означава, че алгоритъмът ще работи мигновено.

Проверяване коректността на новия алгоритъм

Изглежда този път нямаме проблем с производителността. Имаме бърз алгоритъм.

Да проверим дали е коректен. За целта си рисуваме на лист хартия някой по-голям и по-сложен пример, в който има много изходи и много пътища и започваме да изпълняваме алгоритъма. Изглежда работи коректно.

След това пробваме с лабиринт без изход. Изглежда алгоритъмът завършва, но не намира изход. Следователно работи коректно.

Пробваме още 2-3 примера и се убеждаваме, че този алгоритъм винаги намира най-краткия път до някой изход и винаги работи бързо, защото обхожда всяка клетка от лабиринта най-много веднъж.

Какви структури от данни да използваме?

С новия алгоритъм обхождаме последователно всички съседни клетки на началната клетка. Можем да ги сложим в някаква структура данни, примерно масив или по-добре списък(или списък от списъци), понеже в масива не можем да добавяме.

След това взимаме списъка с достигнатите на последната стъпка клетки и добавяме в друг списък техните съседи.

Така, ако индексираме списъците, получаваме списък$_0$, който съдържа началната клетка, списък$_1$, който съдържа проходимите съседни на началната клетка, след това списък$_2$, който съдържа проходимите съседи на списък$_1$ и т.н. На **n**-тата стъпка получаваме списък$_n$, който съдържа всички клетки, достижими за точно **n** стъпки, т.е. клетките на разстояние **n** от стартовата клетка.

Изглежда можем да ползваме списък от списъци, за да пазим клетките, получени на всяка стъпка. Ако се замислим, за да получим **n**-тия списък, ни е достатъчен (**n-1**)-вия. Реално не ни трябва списък от списъци, а само списъкът от последната стъпка.

Можем да достигнем и до по-генерален извод: клетките се обработват в реда на постъпване: когато свършват клетките от стъпка `k`, чак тогава се обработват клетките от стъпка `k+1`, а едва след тях – клетките от стъпка `k+2` и т.н. Процесът прилича на опашка – по-рано постъпилите клетки се обработват по-рано.

За да реализираме алгоритъма, можем да използваме опашка от клетки. За целта трябва да дефинираме клас клетка (`Cell`), който да съдържа координатите на дадена клетка (ред и колона). Можем да пазим в матрицата за всяка клетка на какво разстояние се намира от началната клетка или -1, ако разстоянието още не е пресметнато.

Ако се замислим още малко, разстоянието от стартовата клетка може да се пази в самата клетка (в класа `Cell`) вместо да се прави специална матрица за разстоянията. Така ще се спести памет.

Вече имаме яснота за структурите данни. Остава да реализираме алгоритъма – стъпка по стъпка.

Стъпка 1 – класът Cell

Можем да започнем от дефиницията на класа `Cell`. Той ще ни трябва, за да запазим стартовата клетка, от която започва търсенето на пътя. За да е по-кратък и прегледен кодът ще използваме автоматични свойства (automatic properties[2] или auto-implemented properties). Благодарение на тях не се налага да декларираме поле за съответното свойство, нито да пишем код за извличане или промяна на стойността на полето. Това се извършва автоматично от компилатора по време на компилацията. Чрез инструментите за дисасемблиране **JustDecompiler** или **ILSpy**, споменати в глава 1, може да проверите какво е генерирал компилаторът при използването на автоматични свойства във вашата програма. Ето го и класът `Cell`:

```
public class Cell
{
  public int Row { get; set; }
  public int Column { get; set; }
  public int Distance { get; set; }
}
```

Може да му добавим и конструктор за удобство:

---

[2] Повече информация за автоматичните свойства можете да прочетете от:
- http://msdn.microsoft.com/en-us/library/bb384054.aspx
- http://csharp.net-tutorials.com/csharp-3.0/automatic-properties/

```
public Cell(int row, int column, int distance)
{
   this.Row = row;
   this.Column = column;
   this.Distance = distance;
}
```

Стъпка 2 – прочитане на входния файл

Ще четем входния файл ред по ред чрез познатия ни клас **`StreamReader`**. На всеки ред ще анализираме символите и ще ги записваме в матрица от символи. При достигане на символ "*" ще запомним координатите му в инстанция на класа **`Cell`**, за да знаем от къде да започнем търсенето на най-краткия път за излизане от лабиринта.

Можем да дефинираме клас **`Maze`** и в него да пазим матрицата на лабиринта и стартовата клетка:

Maze.cs

```
public class Maze
{
   private char[,] maze;
   private int size;
   private Cell startCell = null;

   public void ReadFromFile(string fileName)
   {
      using (StreamReader reader = new StreamReader(fileName))
      {
         // Read maze size and create maze
         this.size = int.Parse(reader.ReadLine());
         this.maze = new char[this.size, this.size];

         // Read the maze cells from the file
         for (int row = 0; row < this.size; row++)
         {
            string line = reader.ReadLine();
            for (int col = 0; col < this.size; col++)
            {
               this.maze[row, col] = line[col];
               if (line[col] == '*')
               {
                  this.startCell = new Cell(row, col, 0);
               }
            }
         }
      }
   }
}
```

За простота ще пропуснем обработката на грешки при четене и писане във файл. При възникване на изключение го изхвърляме от главния метод и ще оставяме CLR да го отпечата в конзолата.

Вече имаме класа **`Maze`** и подходящо представяне на данните от входния файл. За да сме сигурни, че написаното дотук е вярно трябва да тестваме. Можем да проверим дали матрицата е вярно попълнена, като я отпечатаме на конзолата. Друг вариант е да разгледаме стойностите на полетата от класа **`Maze`** през дебъгера на Visual Studio.

След като тестваме написаното дотук продължаваме със следващата стъпка, а именно търсенето на най-краткия път.

Стъпка 3 – намиране на най-къс път

Можем директно да имплементираме алгоритъма, който вече дискутирахме. Трябва да дефинираме опашка и в нея да сложим в началото стартовата клетка. След това в цикъл трябва да взимаме поредната клетка от опашката и да добавяме всичките ѝ непосетени проходими съседи. На всяка стъпка има шанс да стъпим в клетка от границата на лабиринта, при което считаме, че сме намерили изход и търсенето приключва. Повтаряме цикъла докато опашката свърши. При всяко посещение на дадена клетка проверяваме дали клетката е свободна и ако е свободна, я маркираме като непроходима. Така избягваме повторно попадане в същата клетка. Ето как изглежда имплементацията на алгоритъма:

```
public int FindShortestPath()
{
   // Queue for traversing the cells in the maze
   Queue<Cell> visitedCells = new Queue<Cell>();
   VisitCell(visitedCells, this.startCell.Row,
      this.startCell.Column, 0);

   // Perform Breath-First-Search (BFS)
   while (visitedCells.Count > 0)
   {
      Cell currentCell = visitedCells.Dequeue();
      int row = currentCell.Row;
      int column = currentCell.Column;
      int distance = currentCell.Distance;
      if ((row == 0) || (row == size - 1)
         || (column == 0) || (column == size - 1))
      {
         // We are at the maze border
         return distance + 1;
```

```
        }
        VisitCell(visitedCells, row, column + 1, distance + 1);
        VisitCell(visitedCells, row, column - 1, distance + 1);
        VisitCell(visitedCells, row + 1, column, distance + 1);
        VisitCell(visitedCells, row - 1, column, distance + 1);
    }

    // We didn't reach any cell at the maze border -> no path
    return -1;
}

private void VisitCell(Queue<Cell> visitedCells,
    int row, int column, int distance)
{
    if (this.maze[row, column] != 'x')
    {
        // The cell is free --> visit it
        maze[row, column] = 'x';
        Cell cell = new Cell(row, column, distance);
        visitedCells.Enqueue(cell);
    }
}
```

Проверка след стъпка 3

Преди да се захванем със следващата стъпка, трябва да тестваме, за да проверим нашия алгоритъм. Трябва да пробваме нормалния случай, както и граничните случаи, когато няма изход, когато се намираме на изход, когато входният файл не съществува или квадратната матрица е с размер нула. Едва след това може да започнем решаването на следващата стъпка.

Нека да пробваме случая, в който имаме дължина нула на квадратната матрица във входния файл:

```
Unhandled   Exception:   System.NullReferenceException:   Object
reference not set to an instance of an object.
 at Maze.FindShortestPath()
```

Допуснали сме грешка. Проблемът е в това, че при създаване на обект от класа **`Maze`**, променливата, в която ще помним началната клетка, се инициализира с **`null`**. Ако лабиринтът няма клетки (дължина 0) или липсва стартовата клетка, би трябвало програмата да връща резултат -1, а не да дава изключение. Можем да добавим проверка в началото на метода **`FindShortestPath()`**:

```
public int FindShortestPath()
{
   if (this.startCell == null)
   {
      // Start cell is missing -> no path
      return -1;
   }
   …
```

В останалите случаи изглежда, че алгоритъмът работи.

Стъпка 4 – записване на резултата във файл

Остава да запишем резултата от метода **`FindShortestWay()`** в изходния файл. Това е тривиална задача:

```
public void SaveResult(String fileName, int result)
{
   using (StreamWriter writer = new StreamWriter(fileName))
   {
      writer.WriteLine("The shortest way is: " + result);
   }
}
```

Ето как изглежда пълният код на решението на задачата:

Maze.cs

```
using System.IO;
using System.Collections.Generic;

public class Maze
{
   private const string InputFileName = "Problem2.in";
   private const string OutputFileName = "Problem2.out";

   public class Cell
   {
      public int Row { get; set; }
      public int Column { get; set; }
      public int Distance { get; set; }

      public Cell(int row, int column, int distance)
      {
         this.Row = row;
         this.Column = column;
         this.Distance = distance;
      }
```

```
}

private char[,] maze;
private int size;
private Cell startCell = null;

public void ReadFromFile(string fileName)
{
  using (StreamReader reader = new StreamReader(fileName))
  {
    // Read maze size and create maze
    this.size = int.Parse(reader.ReadLine());
    this.maze = new char[this.size, this.size];

    // Read the maze cells from the file
    for (int row = 0; row < this.size; row++)
    {
      string line = reader.ReadLine();
      for (int col = 0; col < this.size; col++)
      {
        this.maze[row, col] = line[col];
        if (line[col] == '*')
        {
          this.startCell = new Cell(row, col, 0);
        }
      }
    }
  }
}

public int FindShortestPath()
{
  if (this.startCell == null)
  {
    // Start cell is missing -> no path
    return -1;
  }

  // Queue for traversing the cells in the maze
  Queue<Cell> visitedCells = new Queue<Cell>();
  VisitCell(visitedCells, this.startCell.Row,
    this.startCell.Column, 0);

  // Perform Breath-First-Search (BFS)
  while (visitedCells.Count > 0)
  {
    Cell currentCell = visitedCells.Dequeue();
    int row = currentCell.Row;
    int column = currentCell.Column;
```

```
        int distance = currentCell.Distance;
        if ((row == 0) || (row == size - 1)
          || (column == 0) || (column == size - 1))
        {
          // We are at the maze border
          return distance + 1;
        }

        VisitCell(visitedCells, row, column + 1, distance + 1);
        VisitCell(visitedCells, row, column - 1, distance + 1);
        VisitCell(visitedCells, row + 1, column, distance + 1);
        VisitCell(visitedCells, row - 1, column, distance + 1);
      }

      // We didn't reach any cell at the maze border -> no path
      return -1;
    }

    private void VisitCell(Queue<Cell> visitedCells,
      int row, int column, int distance)
    {
      if (this.maze[row, column] != 'x')
      {
        // The cell is free --> visit it
        maze[row, column] = 'x';
        Cell cell = new Cell(row, column, distance);
        visitedCells.Enqueue(cell);
      }
    }

    public void SaveResult(String fileName, int result)
    {
      using (StreamWriter writer = new StreamWriter(fileName))
      {
        writer.WriteLine("The shortest way is: " + result);
      }
    }

    public static void Main()
    {
      Maze maze = new Maze();
      maze.ReadFromFile(InputFileName);
      int pathLength = maze.FindShortestPath();
      maze.SaveResult(OutputFileName, pathLength);
    }
}
```

Тестване на решението на задачата

След като имаме решение на задачата трябва да тестваме. Вече тествахме граничните случаи и случаи като липса на изход или началната позиция да е на изхода. Видяхме, че алгоритъмът работи коректно.

Остава да тестваме с голям лабиринт, например 1000 на 1000. Можем да си направим такъв лабиринт много лесно – с copy/paste. Изпълняваме теста и се убеждаваме, че програмата работи коректно за големия тест и работи изключително бързо – не се усеща каквото и да е забавяне.

При тестването трябва да се опитваме по всякакъв начин да счупим нашето решение. Пускаме още няколко по-трудни примера (примерно лабиринт с проходими клетки във формата на спирала). Можем да сложим голям лабиринт с много пътища, но без изход. Можем да сложим и каквото още се сетим.

Накрая се убеждаваме, че имаме коректно решение и преминаваме към следващата задача.

Задача 3: Магазин за авточасти

Фирма планира създаване на система за управление на магазин за авточасти. Една част може да се използва при различни модели автомобили и има следните характеристики:

Код, наименование, категория (за ходовата част, гуми и джанти, за двигателя, аксесоари и т.н.), покупна цена, продажна цена, списък с модели автомобили, за които може да се използва (даден автомобил се описва с марка, модел и година на производство, примерно Mercedes C320, 2008), фирма-производител.

Фирмите-производители се описват с наименование, държава, адрес, телефон и факс.

Да се проектира съвкупност от класове с връзки между тях, които моделират данните за магазина. Да се напише демонстрационна програма, която показва коректната работа на всички класове.

Измисляне на идея за решение

От нас се изисква да създадем съвкупност от класове и връзки между тях, които да описват данните за магазина. Трябва да разберем кои съществителни са важни за решаването на задачата. Те са обекти от реалния свят, на които съответстват класове.

Кои са тези съществителни, които ни интересуват? Имаме магазин, авточасти, автомобили и фирми-производители. Трябва да създадем клас описващ магазин. Той ще се казва **`Shop`**. Другите класове съответно са **`Part`**, **`Car`** и **`Manufacturer`**. В условието на задачата има и други съществителни, например код на една част или година на производство на дадена кола. За тези съществителни няма да създаваме отделни класове, а вместо това ще бъдат полета в създадените от нас класове. Например в класа **`Part`** ще има примерно поле **`code`** от тип **`string`**.

Вече знаем кои ще са нашите класове, както и полетата, които ги описват. Остава да си изясним връзките между обектите.

Каква структури от данни да използване, за да опишем връзката между два класа?

За да опишем връзката между два класа можем да използваме масив. При масива имаме достъп до елементите му по индекс, но веднъж след като го създадем не можем да му променяме дължината. Това го прави неудобен за нашата задача, понеже не знаем колко части ще имаме в магазина и по всяко време може да докарат още части или някой да купи някоя част и да се наложи да я изтрием или променим.

По-удобен е **`List<T>`**. Той притежава предимствата на масив, а освен това е с променлива дължина и с него лесно се реализира въвеждане и изтриване на елементи.

Засега изглежда, че **`List<T>`** е най-подходящ. За да се убедим ще разгледаме още няколко структури от данни. Например хеш-таблица – не е удобна в този случаи, понеже структурата "части" не от типа ключ-стойност. Тя би била подходяща, ако в магазина всяка част има уникален номер (например баркод). Тогава ще можем да ги търсим по този уникален номер. Структури като стек и опашка са неуместни.

Структурата "множество" и нейната имплементация **`HashSet`** се ползва, когато имаме уникалност по даден ключ. Може би на места ще е добра да ползваме тази структура, за да избегнем повторения. Трябва да имаме предвид, че ползването на **`HashSet<T>`** изисква да имаме методи **`GetHashCode()`** и **`Equals()`**, дефинирани коректно в типа **`T`**.

В крайна сметка избираме да ползваме **`List<T>`** и **`HashSet<T>`**.

Разделяне на задачата на подзадачи

Сега остава да си изясним въпроса от къде да започнем написването на задачата. Ако започнем да пишем класа **`Shop`**, ще се нуждаем от класа **`Part`**. Това ни подсеща, че трябва да започнем от клас, който не зависи от другите. Ще разделим написването на всеки клас на подзадача, като ще започнем от независещите от другите класове:

- Клас описващ автомобил – **`Car`**
- Клас описващ производител на части – **`Manufacturer`**
- Клас описващ част за автомобили – **`Part`**
- Клас за магазина – **`Shop`**

- Клас за тестване на останалите класове с примерни данни – **TestShop**

Имплементиране: стъпка по стъпка

Започваме написването на класовете, които сме описали в нашата идея. Ще ги създаваме в реда, по който са изброени в списъка.

Стъпка 1: класът Car

Започваме решаването на задачата с дефинирането на класа **Car**. В дефиницията имаме три полета, които показват производителя, модела и годината на производство на една кола и стандартния метод **ToString()**, който връща низ с информация за дадена кола. Дефинираме го по следния начин:

Car.cs

```
public class Car
{
   private string brand;
   private string model;
   private string productionYear;

   public Car(string brand, string model, string productionYear)
   {
      this.brand = brand;
      this.model = model;
      this.productionYear = productionYear;
   }

   public override string ToString()
   {
      return "<" + this.brand + "," + this.model + ","
         + this.productionYear + ">";
   }
}
```

Стъпка 2: класът Manufacturer

Следва да реализираме дефиницията на класа **Manufacturer**, който описва производителя на дадена част. Той ще има пет полета – име, държава, адрес, телефонен номер и факс. Ще предефинираме стандартния метод **ToString()**, с който ще представяме цялата информацията за дадена инстанция на класа **Manufacturer**.

Manufacturer.cs

```
public class Manufacturer
```

```
{
   private string name;
   private string country;
   private string address;
   private string phoneNumber;
   private string fax;

   public Manufacturer(string name, string country,
      string address, string phoneNumber, string fax)
   {
      this.name = name;
      this.country = country;
      this.address = address;
      this.phoneNumber = phoneNumber;
      this.fax = fax;
   }

   public override string ToString()
   {
      return this.name + " <" + this.country + "," + this.address
         + "," + this.phoneNumber + "," + this.fax + ">";
   }
}
```

Стъпка 3: класът Part

Сега трябва да дефинираме класа **Part**. Дефиницията му ще включва следните полета – име, код, категория, списък с коли, с които може да се използва дадената част, начална и крайна цена и производител. Тук вече ще използваме избраната от нас структура от данни **HashSet<T>**. В случая ще бъде **HashSet<Car>**. Полето показващо производителя на частта ще бъде от тип **Manufacturer**, защото задача изисква да се помни допълнителна информация за производителя. Ако се искаше да се знае само името на производителя (както случая с класа Car) нямаше да има нужда от този клас. Щяхме да имаме поле от тип **string**. За полето, което описва категорията на частта ще използваме **enum**:

PartCategory.cs

```
public enum PartCategory
{
   Engine,
   Tires,
   Exhaust,
   Suspention,
   Brakes
}
```

Нужен ни е метод за добавяне на кола (обект от тип **`Car`**) в списъка с колите (в **`HashSet<Car>`**). Той ще се казва **`AddSupportedCar(Car car)`**. Ето го и кода на класа **`Part`**:

Part.cs

```
public class Part
{
  private String name;
  private String code;
  private PartCategory category;
  private HashSet<Car> supportedCars;
  private double buyPrice;
  private double sellPrice;
  private Manufacturer manufacturer;

  public Part(string name, double buyPrice, double sellPrice,
    Manufacturer manufacturer, string code,
    PartCategory category)
  {
    this.name = name;
    this.buyPrice = buyPrice;
    this.sellPrice = sellPrice;
    this.manufacturer = manufacturer;
    this.code = code;
    this.category = category;
    this.supportedCars = new HashSet<Car>();
  }

  public void AddSupportedCar(Car car)
  {
    this.supportedCars.Add(car);
  }

  public override string ToString()
  {
    StringBuilder result = new StringBuilder();
    result.Append("Part: " + this.name + "\n");
    result.Append("-code: " + this.code + "\n");
    result.Append("-category: " + this.category + "\n");
    result.Append("-buyPrice: " + this.buyPrice + "\n");
    result.Append("-sellPrice: " + this.sellPrice + "\n");
    result.Append("-manufacturer: " +
      this.manufacturer + "\n");
    result.Append("---Supported cars---" + "\n");
    foreach (Car car in this.supportedCars)
    {
      result.Append(car);
      result.Append("\n");
    }
    result.Append("----------------------\n");
    return result.ToString();
  }
}
```

Понеже в **Part** ползваме **HashSet<Car>** е необходимо да предефинираме методите **GetHashCode()** и **Equals()** за класа **Car**:

```
public override int GetHashCode()
{
  const int prime = 31;
  int result = 1;
  result = prime * result + ((this.brand == null) ? 0 :
    this.brand.GetHashCode());
  result = prime * result + ((this.model == null) ? 0 :
    this.model.GetHashCode());
  result = prime * result + ((this.productionYear == null) ? 0 :
    this.productionYear.GetHashCode());
  return result;
}

public override bool Equals(Object obj)
{
  if (this == obj)
  {
    return true;
  }

  if (obj == null)
  {
    return false;
  }

  if (this.GetType() != obj.GetType())
  {
    return false;
  }

  Car other = (Car)obj;
  if (this.brand == null)
  {
    if (other.brand != null)
    {
      return false;
    }
  }
  else if (!this.brand.Equals(other.brand))
  {
      return false;
   }

   if (this.model == null)
   {
      if (other.model != null)
      {
         return false;
      }
   }
   else if (!this.model.Equals(other.model))
   {
      return false;
   }

   if (this.productionYear == null)
   {
      if (other.productionYear != null)
      {
         return false;
      }
   }
   else if (!this.productionYear.Equals(other.productionYear))
   {
      return false;
   }

   return true;
}
```

Стъпка 4: класът Shop

Вече имаме всички нужни класове за създаване на класа **`Shop`**. Той ще има две полета – име и списък от части, които се продават. Списъкът ще бъде **`List<Part>`**. Ще добавим метода **`AddPart(Part part)`**, чрез който ще добавяме нова част. С предефинирания **`ToString()`** ще отпечатваме името на магазина и частите в него. Ето примерна реализация:

Shop.cs

```
public class Shop
{
   private string name;
   private List<Part> parts;

   public Shop(string name)
   {
      this.name = name;
```

```
        this.parts = new List<Part>();
    }

    public void AddPart(Part part)
    {
        this.parts.Add(part);
    }

    public override string ToString()
    {
        StringBuilder result = new StringBuilder();
        result.Append("Shop: " + this.name + "\n\n");
        foreach (Part part in this.parts)
        {
            result.Append(part);
            result.Append("\n");
        }
        return result.ToString();
    }
}
```

Стъпка 5: класът TestShop

Създадохме всички нужни класове. Остава да създадем още един, с който да демонстрираме използването на всички останали класове. Той ще се казва **`TestShop`**. В **`Main()`** метода ще създадем два производителя и няколко коли. Ще ги добавим към две части. Частите ще добавим към обект от тип **`Shop`**. Накрая ще отпечатаме всичко на конзолата. Ето примерния код:

TestShop.cs

```
public class TestShop
{
    public static void Main()
    {
        Manufacturer bmw = new Manufacturer("BWM",
            "Germany", "Bavaria", "665544", "876666");
        Manufacturer lada = new Manufacturer("Lada",
            "Russia", "Moscow", "653443", "893321");

        Car bmw316i = new Car("BMW", "316i", "1994");
        Car ladaSamara = new Car("Lada", "Samara", "1987");
        Car mazdaMX5 = new Car("Mazda", "MX5", "1999");
        Car mercedesC500 = new Car("Mercedes", "C500", "2008");
        Car trabant = new Car("Trabant", "super", "1966");
        Car opelAstra = new Car("Opel", "Astra", "1997");
        Part cheapPart = new Part("Tires 165/50/13", 302.36,
          345.58, lada, "T332", PartCategory.Tires);
        cheapPart.AddSupportedCar(ladaSamara);
        cheapPart.AddSupportedCar(trabant);

        Part expensivePart = new Part("BMW Engine Oil",
          633.17, 670.0, bmw, "Oil431", PartCategory.Engine);
        expensivePart.AddSupportedCar(bmw316i);
        expensivePart.AddSupportedCar(mazdaMX5);
        expensivePart.AddSupportedCar(mercedesC500);
        expensivePart.AddSupportedCar(opelAstra);

        Shop newShop = new Shop("Tunning shop");
        newShop.AddPart(cheapPart);
        newShop.AddPart(expensivePart);

        Console.WriteLine(newShop);
    }
}
```

Това е резултатът от изпълнението на нашата програма:

```
Shop: Tunning shop

Part: Tires 165/50/13
-code: T332
-category: TIRES
-buyPrice: 302.36
-sellPrice: 345.58
-manufacturer: Lada <Russia,Moscow,653443,893321>
---Supported cars---
<Lada,Samara,1987>
<Trabant,super,1966>
----------------------

Part: BMW Engine Oil
-code: Oil431
-category: ENGINE
-buyPrice: 633.17
-sellPrice: 670.0
-manufacturer: BWM <Germany,Bavaria,665544,876666>
---Supported cars---
<Opel,Astra,1997>
<BMW,316i,1994>
<Mazda,MX5,1999>
<Mercedes,C500,2008>
----------------------
```

Тестване на решението

Накрая остава да тестваме нашата задача. Всъщност ние направихме това с класа **`TestShop`**. Това обаче не означава, че сме изтествали напълно нашата задача. Трябва да се проверят граничните случаи, например когато някои от списъците са празни. Да променим малко кода в **`Main()`** метода, за да пуснем задачата с празен списък:

TestShop.cs

```
public class TestShop
{
   public static void Main()
   {
      Manufacturer bmw = new Manufacturer("BWM",
         "Germany", "Bavaria", "665544", "876666");
      Manufacturer lada = new Manufacturer("Lada",
         "Russia", "Moscow", "653443", "893321");

      Car bmw316i = new Car("BMW", "316i", "1994");
      Car ladaSamara = new Car("Lada", "Samara", "1987");
      Car mazdaMX5 = new Car("Mazda", "MX5", "1999");
      Car mercedesC500 = new Car("Mercedes", "C500", "2008");
      Car trabant = new Car("Trabant", "super", "1966");
      Car opelAstra = new Car("Opel", "Astra", "1997");

      Part cheapPart = new Part("Tires 165/50/13", 302.36,
         345.58, lada, "T332", PartCategory.Tires);

      Part expensivePart = new Part("BMW Engine Oil",
         633.17, 670.0, bmw, "Oil431", PartCategory.Engine);
      expensivePart.AddSupportedCar(bmw316i);
      expensivePart.AddSupportedCar(mazdaMX5);
      expensivePart.AddSupportedCar(mercedesC500);
      expensivePart.AddSupportedCar(opelAstra);

      Shop newShop = new Shop("Tunning shop");
      newShop.AddPart(cheapPart);
      newShop.AddPart(expensivePart);

      Console.WriteLine (newShop);
   }
}
```

Резултатът от този тест е следният:

```
Shop: Tunning shop

Part: Tires 165/50/13
```

```
-code: T332
-category: TIRES
-buyPrice: 302.36
-sellPrice: 345.58
-manufacturer: Lada <Russia,Moscow,653443,893321>
---Supported cars---
----------------------

Part: BMW Engine Oil
-code: Oil431
-category: ENGINE
-buyPrice: 633.17
-sellPrice: 670.0
-manufacturer: BWM <Germany,Bavaria,665544,876666>
---Supported cars---
<Opel,Astra,1997>
<BMW,316i,1994>
<Mazda,MX5,1999>
<Mercedes,C500,2008>
----------------------
```

От резултата се вижда, че списъкът от коли на евтината част е празен. Това е и правилният изход. Следователно нашата задача изпълнява коректно граничния случай с празен списък.

Упражнения

1. Даден входен файл **`mails.txt`**, който съдържа имена на потребители и техните email адреси. Всеки ред от файла изглежда така:

   ```
   <first name> <last name> <username>@<host>.<domain>
   ```

   Има изискване за имейл адресите – **`<username>`** може да е последователност от латински букви (**`a-z`**, **`A-Z`**) и долна черна (**`_`**), **`<host>`** е последователност от малки латински букви (**`a-z`**), а **`<domain>`** има ограничение от 2 до 4 малки латински букви (**`a-z`**). Да се напише програма, която намира валидните email адреси и ги записва заедно с имената на потребителите в изходен файл **`validMails.txt`**.

2. Даден е лабиринт, който се състои от N x N квадратчета, всяко от които може да е проходимо (0) или не (x).

   В едно от квадратчетата се намира отново нашият герой Минчо (*). Две квадратчета са съседни, ако имат обща стена. Минчо може на една стъпка да преминава от едно проходимо квадратче в съседно на него проходимо квадратче. Напишете програма, която по даден лабиринт отпечатва броя на възможните изходи от лабиринта.

x	x	x	0	x	x
0	x	0	0	0	
0	*	0	x	0	0
x	x	x	x	0	x
0	0	0	0	0	x
0	x	0	x	x	0

Входните данни се четат от текстов файл с име `Problem.in`. На първия ред във файла стои числото N (2 < N < 1000). На следващите N реда стоят по N символа, всеки от които е или "`0`" или "`x`" или "`*`". Изходът представлява едно число и трябва да се изведе във файла `Problem.out`.

3. Даден е лабиринт, който се състои от N x N квадратчета, всяко от които може да е проходимо или не. Проходимите клетки съдържат малка латинска буква между "`a`" и "`z`", а непроходимите – '`#`'. В едно от квадратчетата се намира Минчо. То е означено с "`*`".

   Две квадратчета са съседни, ако имат обща стена. Минчо може на една стъпка да преминава от едно проходимо квадратче в съседно на него проходимо квадратче. Когато Минчо минава през проходимите квадратчета, той си записва буквите от всяко квадратче. На всеки изход получава дума. Напишете програма, която по даден лабиринт отпечатва думите, които се образуват при всички възможни изходи от лабиринта.

a	#	#	k	m	#
z	#	a	d	a	#
a	*	m	#	#	#
#	d	#	#	#	#
r	i	f	i	d	#
#	d	#	d	#	t

   Входните данни се четат от текстов файл с име `Problem.in`. На първия ред във файла стои числото N (2 < N < 10). На следващите N реда стоят по N символа, всеки от които е или латинска буква между "`a`" и "`z`" или "`#`" или "`*`". Изходът трябва да се изведе във файла `Problem.out`.

4. Фирма планира създаване на система за управление на звукозаписна компания. Звукозаписната компания има име, адрес, собственик и изпълнители. Всеки изпълнител има име, псевдоним и създадени албуми. Албумите се описват с име, жанр, година на издаване, брой на продадените копия и списък от песни. Песните, от своя страна се описват с

име и времетраене. Да се проектира съвкупност от класове с връзки между тях, които моделират данните за звукозаписната компания. Да се реализира тестов клас, който демонстрира работата на всички останали класове.

5. Фирма планира създаване на система за управление на компания за недвижими имоти. Компанията има име, собственик, Булстат, служители и разполага със списък от имоти за продажба. Служители се описват с име, длъжност и стаж. Компанията продава няколко вида имоти – апартаменти, къщи, незастроени площи и магазини. Всички те се характеризират с площ, цена на квадратен метър и местоположение. За някои от тях има допълнителна информация. За апартамента има данни за номер на етажа, дали в блока има асансьор и дали е обзаведен. За къщите се знаят квадратните метри на застроена част и на незастроената (двора), на колко етажа е и дали е обзаведена. Да се проектира съвкупност от класове с връзки между тях, които моделират данните за компанията. Да се реализира тестов клас, който демонстрира работата на всички останали класове.

Решения и упътвания

1. Задачата е подобна на първата от примерния изпит. Отново трябва да чете ред по ред от входния файл и чрез подходящ регулярен израз да извличате имейл адресите.

   Примерен входен файл:

```
Ivan Dimitrov ivan_dimitrov@abv.bg
Svetlana Todorova Svetlana_tv@mail.bg
Kiril Kalchev kalchev@gmail.com
Todor Ivanov todo*r@888.com
Ivelina Petrova ivel&7@abv.bg
Petar Petrov pesho<5.mail.bg
```

   Изходен файл:

```
Ivan Dimitrov ivan_dimitrov@abv.bg
Svetlana Todorova Svetlana_tv@mail.bg
Kiril Kalchev kalchev@gmail.com
```

   Тествайте внимателно решението си преди да преминете към следващата задача.

2. Възможните изходи от лабиринта са всички клетки, които се намират на границата на лабиринта и са достижими от стартовата клетка. Задачата се решава с дребна модификация на решението на задачата за лабиринта.

3. Задачата е изглежда подобна на предната, но се искат всички възможни пътища до изхода. Можете да направите рекурсивно търсене

с връщане назад (backtracking) и да натрупвате в **`StringBuilder`** буквите до изхода, за да образувате думите, които трябва да се отпечатат. При големи лабиринти задачата няма добро решение (защото се използва пълно изчерпване и броят пътища до някой от изходите може да е ужасно голям).

4. Трябва да напишете нужните класове – **`MusicCompany`**, **`Singer`**, **`Album`**, **`Song`**. Помислете за връзките между класовете и какви структури данни да ползвате за тях. За отпечатването предефинирайте метода **`ToString()`** от **`System.Object`**. Тествайте всички методи и граничните случаи.

5. Класовете, които трябва да напишете са **`EstateCompany`**, **`Employee`**, **`Apartment`**, **`House`**, **`Shop`** и **`Area`**. Забележете, че класовете, които ще описват недвижимите имоти имат някои еднакви характеристики. Изнесете тези характеристики в базов отделен клас **`Estate`**. Създайте метод **`ToString()`**, който да изписва на конзолата данните от този клас. Пренапишете метода за класовете, които наследяват този клас, за да показва цялата информация за всеки клас. Тествайте всички методи и граничните случаи.

Telerik Academy

Присъедини се към Академията на Телерик!

АКАДЕМИЯТА НА ТЕЛЕРИК предоставя безплатно практическо обучение, насочено към всички млади хора, желаещи да станат умели .NET софтуерни инженери и да се присъединят към екипа на Телерик.

Академията подготвя бъдещите софтуерни специалисти за сложността, разнообразието и динамиката на днешната ИТ действителност, като за целта всички курсисти се обучават да работят с най-новите технологии на Майкрософт и черпят знания и опит от доказали се в областта на програмирането лектори и разработчици на софтуер.

В академията ще получите задълбочени знания и опит, изучавайки:

C#, .NET технологии, бази данни, SQL, Silverlight, WPF, ASP.NET, HTML5, разработка на мобилни приложения за iOS, Android и Windows Phone, основите на софтуерното инженерство

Академията на Телерик ви дава възможност да:

- **Учите напълно БЕЗПЛАТНО**
- **Изберете сред редица РАЗЛИЧНИ КУРСОВЕ**
- **Овладеете ОСНОВИТЕ на софтуерното инженерство**
- **Усвоите ПРОЦЕСА за разработка на софтуер**
- **Получите задълбочени теоретични и практически ИТ ПОЗНАНИЯ**
- **Станете умел .NET СОФТУЕРЕН ИНЖЕНЕР**
- **Започнете своята ИТ кариера в ТЕЛЕРИК – РАБОТОДАТЕЛ #1 в България за 2010 г.**

Само в рамките на две години **АКАДЕМИЯТА НА ТЕЛЕРИК** за софтуерни инженери успя да се наложи като безспорен лидер у нас в предлагането на допълнително обучение за софтуерни специалисти, спомагайки за успешния старт в кариерното развитие на стотици ентусиазирани младежи.

academy.telerik.com

academy@telerik.com *facebook.com/TelerikAcademy*

telerik

deliver more than expected

Глава 25. Практически задачи за изпит по програмиране – тема 2

В тази тема...

В настоящата тема ще разгледаме условията и ще предложим решения на няколко практически алгоритмични задачи от примерен изпит по програмиране. При решаването на задачите ще се придържаме към съветите от темата "Как да решаваме задачи по програмиране" и ще онагледим прилагането им в практиката.

Задача 1: Броене на думи в текст

Напишете програма, която преброява думите в даден текст, въведен от конзолата. Програмата трябва да извежда общия брой думи, броя думи, изписани изцяло с главни букви и броя думи, изписани изцяло с малки букви. Ако дадена дума се среща няколко пъти на различни места в текста, всяко срещане се брои като отделна дума. За разделител между думите се счита всеки символ, който не е буква.

Примерен вход:

```
Добре дошли на вашия първи изпит по програмиране! Можете ли да
измислите и напишете решение на тази задача? УСПЕХ!
```

Примерен изход:

```
Общо думи: 19
Думи с главни букви: 1
Думи с малки букви: 16
```

Намиране на подходяща идея за решение

Интуитивно ни идва наум, че можем да решим задачата, като разделим текста на отделни думи и след това преброим тези, които ни интересуват.

Тази идея очевидно е вярна, но е прекалено обща и не ни дава конкретен метод за решаването на проблема. Нека се опитаме да я конкретизираме и да проверим дали е възможно чрез нея да реализираме алгоритъм, който да доведе до решение на задачата. Може да се окаже, че реализацията е трудна или сложността на решението е прекалено голяма и нашата програма няма да може да завърши своето изпълнение, дори и с помощта на съвременните мощни компютри. Ако това се случи, ще се наложи да потърсим друго решение на задачата.

Разбиване на задачата на подзадачи

Полезен подход при решаването на алгоритмични задачи е да се опитаме да разбием задачите на подзадачи, които са по-лесно и бързо решими. Нека се опитаме да дефинираме стъпките, които са ни необходими, за решаването на проблема.

Най-напред трябва да разделим текста на отделни думи. Това, само по себе си, не е проста стъпка, но е първата ни крачка към разделянето на проблема на по-малки, макар и все още сложни подзадачи.

Следва преброяване на интересуващите ни думи. Това е втората голяма подзадача, която трябва да решим. Да разгледаме двата проблема по отделно и да се опитаме да ги раздробим на още по-прости задачи.

Как да разделим текста на отделни думи?

За да разделим текста на отделни думи, първо трябва да намерим начин да ги идентифицираме. В условието е казано, че за разделител се счита всеки символ, който не е буква. Следователно първо трябва да идентифицираме разделителите и след това да ги използваме за разделянето на текста на думи.

Ето, че се появиха още две подзадачи – намиране на разделителите в текста и разделяне на текста на думи спрямо разделителите. Решения на тези подзадачи можем да реализираме директно. Това беше и нашата първоначална цел – да разбием сложните задачи на по-малки и лесни подзадачи.

За намиране на разделителите е достатъчно да обходим всички символи и да извлечем тези, които не са букви.

След като имаме разделителите, можем да реализираме разделянето на текста на думи чрез метода **`Split(…)`** на класа **`String`**.

Как да броим думите?

Да предположим, че вече имаме списък с всички думи от текста. Искаме да намерим броя на всички думи на тези, изписани само с главни букви, и на тези, изписани само с малки букви.

За целта можем да обходим всяка дума от списъка и да проверим дали отговаря на някое от условията, които ни интересуват. На всяка стъпка увеличаваме броя на всички думи. Проверяваме дали текущата дума е изписана само с главни букви и, ако това е така, увеличаваме броя на думите с главни букви. Аналогично правим проверка и дали думата е изписана само с малки букви.

Така се появяват още две подзадачи – проверка дали дума е изписана само с главни букви и проверка дали е изписана само с малки букви? Те изглеждат доста лесни. Може би дори е възможно класът **`string`** да ни предоставя наготово такава функционалност. Проверяваме, но се оказва, че не е така. Все пак забелязваме, че има методи, които ни позволяват да преобразуваме символен низ в такъв, съставен само от главни или само от малки букви. Това може да ни помогне.

За да проверим дали една дума е съставена само от главни букви, е достатъчно да сравним думата с низа, който се получава, след като я преобразуваме в дума, съставена само от главни букви. Ако са еднакви, значи резултатът от проверката е истина. Аналогична е и проверката за малките букви.

Проверка на идеята

Изглежда, че идеята ни е добра. Разбихме задачата на подзадачи и знаем как да решим всяка една от тях. Дали да не преминем към имплементацията? Пропуснахме ли нещо?

Не трябваше ли да проверим идеята, разписвайки няколко примера на хартия? Вероятно ще намерим нещо, което сме пропуснали? Можем да започнем с примера от условието:

> Добре дошли на вашия първи изпит по програмиране! Можете ли да измислите и напишете решение на тази задача? УСПЕХ!

Разделителите ще са: **`интервали, ? и !`**. За думите получаваме: **`Добре, дошли, на, вашия, първи, изпит, по, програмиране, Можете, ли, да, измислите, и, напишете, решение, на, тази, задача, УСПЕХ`**.

Преброяваме думите и получаваме коректен резултат. Изглежда идеята е добра и работи. Можем да пристъпим към реализацията. За целта ще имплементираме алгоритъма стъпка по стъпка, като на всяка стъпка ще реализираме по една подзадача.

Да помислим за структурите от данни

Задачата е проста и няма нужда от кой знае какви сложни структури от данни.

За разделителите в текста можем да използваме типа **`char`**. При намирането им ще генерираме един списък с всички символи, които определим за разделители. Можем да използваме **`char[]`** или **`List<char>`**. В случая ще предпочетем втория вариант.

За думите от текста можем да използваме масив от низове **`string[]`** или **`List<string>`**.

Да помислим за ефективността

Има ли изисквания за ефективност? Колко най-дълъг може да е текстът? Тъй като текстът се въвежда от конзолата, той едва ли ще е много дълъг. Никой няма да въведе 1 MB текст от конзолата. Можем да приемем, че ефективността на решението в случая не е застрашена.

Да разпишем на хартия решението на задачата

Много добра стратегия е да се разписва решението на задачата на хартия преди да се започне писането му на компютър. Това помага за откриване отрано на проблеми в идеята или реализацията. Писането на самото решение после става доста по-бързо, защото имаме нещо разписано, а и мозъкът ни е асимилирал добре задачата и нейното решение.

Стъпка 1 – Намиране на разделителите в текста

Ще дефинираме метод, който извлича от текста всички символи, които не са букви, и ги връща в масив от символи, който след това можем да използваме за разделяне на текста на отделни думи:

```
private static char[] ExtractSeparators(string text)
{
   List<char> separators = new List<char>();
   foreach (char character in text)
   {
      // If the character is not a letter,
      // by our definition it is a separator
      if (!char.IsLetter(character))
      {
         separators.Add(character);
      }
   }
   return separators.ToArray();
}
```

Използваме списък от символи **`List<char>`**, където добавяме всички символи, които по нашата дефиниция са разделители в текста.

В цикъл обхождаме всеки един от символите в текста. С помощта на метода **`IsLetter(…)`** на примитивния тип char определяме дали текущия символ е буква и, ако не е, го добавяме към разделителите.

Накрая връщаме масив, съдържащ разделителите.

Изпробване на метода ExtractSeparators(…)

Преди да продължим нататък е редно да изпробваме дали намирането на разделителите работи коректно. За целта, ще си напишем два нови метода. Първият – **`TestExtractSeparators()`**, който ще тества извикването на метода **`ExtractSeparators(…)`**, а вторият – **`GetTestData()`**, който ще ни връща няколко различни текста, с които ще можем да тестваме нашето решение:

```
private static void TestExtractSeparators()
{
   List<string> testData = GetTestData();
   foreach (string testCase in testData)
   {
      Console.WriteLine("Test Case:{0}{1}",
                                Environment.NewLine, testCase);
      Console.WriteLine("Result:");
      foreach (char separator in ExtractSeparators(testCase))
      {
         Console.Write("{0} ", separator);
```

```
    }
    Console.WriteLine();
  }
}

private static List<string> GetTestData()
{
  List<string> testData = new List<string>();
  testData.Add(String.Format("{0}{1}",
          "This is wonderful!!! All separators like ",
          "these ,.(? and these /* are recognized. It works."));
  testData.Add("SingleWord");
  testData.Add(string.Empty);
  testData.Add(">?!>?#@?");
  return testData;
}

static void Main()
{
  string text = "This is wonderful!!! All separators like " +
          "these ,.(? and these /* are recognized. It works.";
  char[] separators = ExtractSeparators(text);
  Console.WriteLine(separators);
}
```

Стартираме програмата и проверяваме дали разделителите са намерени коректно. Резултатът от първият тест е следният:

```
  !!!   ,.(?   /*
```

Изпробваме метода и в някои от граничните случаи – текст, състоящ се от една дума без разделители; текст, съставен само от разделители; празен низ. Всички тези тестове сме добавили в нашия метод **`GetTestData()`**. Изглежда, че методът работи и можем да продължим към реализацията на следващата стъпка.

Стъпка 2 – Разделяне на текста на думи

За разделянето на текста на отделни думи ще използваме разделителите и с помощта на метода **`Split(…)`** на класа **`string`** ще извършим разделянето.

Ето как изглежда нашият метод:

```
private static string[] ExtractWords(string text)
{
  char[] separators = ExtractSeparators(text);
  List<string> extractedWords = new List<string>();
  foreach(string extractedWord in text.Split(separators))
```

```
   {
      // if the word is not empty add it to the
      // extracted words array
      if (!string.IsNullOrEmpty(extractedWord))
      {
         extractedWords.Add(extractedWord);
      }
   }
   return extractedWords.ToArray();
}
```

Преди да преминем към следващата стъпка остава да проверим дали методът работи коректно. За целта ще преизползваме вече написания метод за тестови данни **`GetTestData()`** и ще изтестваме новия метод **`ExtractWords(…)`**:

```
private static void TestExtractWords()
{
   List<string> testData = GetTestData();
   foreach (string testCase in testData)
   {
      Console.WriteLine("Test Case:{0}{1}",
         Environment.NewLine, testCase);
      Console.WriteLine("Result:");
      foreach(string word in ExtractWords(testCase))
      {
         Console.Write("{0} ", word);
      }
   }
}
```

Резултатът от първия тест:

```
This is wonderful All separators like these and these are
recognized IT works
```

Проверяваме резултатите и от другите тестови случаи и се уверяваме, че до тук всичко е вярно и нашият алгоритъм е правилно написан.

Стъпка 3 – Определяне дали дума е изписана изцяло с главни или изцяло с малки букви

Вече имаме идея как да имплементираме тези проверки и можем директно да реализираме методите:

```
private static bool IsUpperCase(string word)
{
   bool result = word.Equals(word.ToUpper());
```

```
    return result;
}

private static bool IsLowerCase(string word)
{
    bool result = word.Equals(word.ToLower());
    return result;
}
```

Изпробваме ги, подавайки им думи, съдържащи само главни, само малки и такива, съдържащи главни и малки букви. Резултатите са коректни.

Стъпка 4 – Преброяване на думите

Вече можем да пристъпим към решаването на проблема – преброяването на думите. Трябва само да обходим списъка с думите и в зависимост каква е думата да увеличим съответните броячи, след което да отпечатаме резултата:

```
private static void CountWords(string[] words)
{
    int allUpperCaseWordsCount = 0;
    int allLowerCaseWordsCount = 0;
    foreach (string word in words)
    {
        if (IsUpperCase(word))
        {
            allUpperCaseWordsCount++;
        }
        else if (IsLowerCase(word))
        {
            allLowerCaseWordsCount++;
        }
    }

    Console.WriteLine("Total words count:{0}", words.Length);
    Console.WriteLine("Upper case words count:{0}",
        allUpperCaseWordsCount);
    Console.WriteLine("Lower case words count:{0}",
        allLowerCaseWordsCount);
}
```

Нека проверим дали броенето работи коректно. Ще си напишем още една тестова функция, използвайки тестовите данни от метода **`GetTestData()`** и вече написания и изтестван от нас метод **`ExtractWords(…)`**:

```
private static void TestCountWords()
{
  List<string> testData = GetTestData();
  foreach (string testCase in testData)
  {
    Console.WriteLine("Test Case:{0}{1}",
      Environment.NewLine, testCase);
    Console.WriteLine("Result:");
    CountWords(ExtractWords(testCase));
  }
}
```

Стартираме приложението и получаваме верен резултат:

```
Total words count: 13
Upper case words count: 1
Lower case words count: 10
```

Проверяваме резултатите и в граничните случаи, когато списъкът съдържа думи само с главни или само с малки букви както и, когато списъкът е празен.

Стъпка 5 – Вход от конзолата

Остава да реализираме и последната стъпка, даваща възможност на потребителя да въвежда текст:

```
private static string ReadText()
{
  Console.WriteLine("Enter text:");
  return Console.ReadLine();
}
```

Стъпка 6 – Сглобяване на всички части в едно цяло

След като сме решили всички подзадачи, можем да пристъпим към пълното решаване на проблема. Остава да добавим **`Main(…)`** метод, в който да съединим отделните парчета:

```
static void Main()
{
  string text = ReadText();
  string[] words = ExtractWords(text);
  CountWords(words);
}
```

Тестване на решението

Докато писахме решението, написахме методи за тестване на всеки един метод, като постепенно интегрирахме методите един с друг. Така в момента сме сигурни, че те работят добре заедно, не сме изпуснали нещо и нямаме метод, който да прави нещо, което не ни е нужно или да дава грешни резултати.

Ако имаме желание да тестваме решението с още данни, достатъчно е само да допишем още данни в метода **`GetTestData(…)`**. Ако искаме, дори можем да модифицираме кода на метода **`GetTestData(…)`**, така че да чете данните за тестване от външен източник – например текстов файл.

Ето как изглежда кодът на цялостното решение:

WordsCounter.cs

```
using System;
using System.Collections.Generic;

public class WordsCounter
{
  static void Main()
  {
    string text = ReadText();
    string[] words = ExtractWords(text);
    CountWords(words);
  }

  private static string ReadText()
  {
    Console.WriteLine("Enter text:");
    return Console.ReadLine();
  }

  private static char[] ExtractSeparators(string text)
  {
    List<char> separators = new List<char>();
    foreach (char character in text)
    {
      // If the character is not a letter, by our
      // definition it is a separator
      if (!char.IsLetter(character))
      {
        separators.Add(character);
      }
    }
    return separators.ToArray();
```

```
   }

   private static string[] ExtractWords(string text)
   {
      char[] separators = ExtractSeparators(text);
      List<string> extractedWords = new List<string>();
      foreach (string extractedWord in
         text.Split(separators.ToArray()))
      {
         // if the word is not empty add it to the extracted
         // words
         if (!string.IsNullOrEmpty(extractedWord))
         {
            extractedWords.Add(extractedWord);
         }
      }

      return extractedWords.ToArray();
   }

   private static bool IsUpperCase(string word)
   {
      bool result = word.Equals(word.ToUpper());
      return result;
   }

   private static bool IsLowerCase(string word)
   {
      bool result = word.Equals(word. ToLower());
      return result;
   }

   private static void CountWords(string[] words)
   {
      int allUpperCaseWordsCount = 0;
      int allLowerCaseWordsCount = 0;

      foreach (string word in words)
      {
         if (IsUpperCase(word))
         {
            allUpperCaseWordsCount++;
         }
         else if (IsLowerCase(word))
         {
            allLowerCaseWordsCount++;
         }
      }
```

```
    Console.WriteLine("Total words count:{0}",
      words.Length));
    Console.WriteLine("Upper case words count: {0}",
      allUpperCaseWordsCount));
    Console.WriteLine("Lower case words count: {0}",
      allLowerCaseWordsCount));
  }

  private static List<string> GetTestData()
  {
    List<string> testData = new List<string>();
    testData.Add(String.Format("{0}{1}",
      "This is wonderful!!! All separators like ",
      "these ,.(? and these /* are recognized. IT works."));
    testData.Add("SingleWord");
    testData.Add(string.Empty);
    testData.Add(">?!>?#@?");
    return testData;
  }

  private static void TestExtractSeparators()
  {
    List<string> testData = GetTestData();
    foreach (string testCase in testData)
    {
      Console.WriteLine("Test Case:{0}{1}",
        Environment.NewLine, testCase);
      Console.WriteLine("Result:");
      foreach (char separator in
        ExtractSeparators(testCase))
      {
        Console.Write("{0} ", separator);
      }
      Console.WriteLine();
    }
  }

  private static void TestExtractWords()
  {
    List<string> testData = GetTestData();

    foreach (string testCase in testData)
    {
      Console.WriteLine("Test Case:{0}{1}",
        Environment.NewLine, testCase);
      Console.WriteLine("Result:");
      foreach (string word in ExtractWords(testCase))
      {
        Console.Write("{0} ", word);
```

```
            }
        }
    }

    private static void TestCountWords()
    {
        List<string> testData = GetTestData();
        foreach (string testCase in testData)
        {
            Console.WriteLine("Test Case:{0}{1}",
                Environment.NewLine, testCase);
            Console.WriteLine("Result:");
            CountWords(ExtractWords(testCase));
        }
    }
}
```

Дискусия за производителността

Тъй като въпросът за производителността в тази задача не е явно поставен, само ще дадем идея как бихме могли да реагираме, ако евентуално се окаже, че нашият алгоритъм е бавен. Понеже разделянето на текста по разделящите символи предполага, че целият текст трябва да бъде прочетен в паметта и думите, получени при разделянето също трябва да се запишат в паметта, то програмата ще консумира голямо количество памет, ако входният текст е голям. Например, ако входът е 200 MB текст, програмата ще изразходва най-малко 800 MB памет, тъй като всяка дума се пази два пъти по 2 байта за всеки символ.

Ако искаме да избегнем консумацията на голямо количество памет, трябва да не пазим всички думи едновременно в паметта. Можем да измислим друг алгоритъм: сканираме текста символ по символ и натрупваме буквите в някакъв буфер (например **`StringBuilder`**). Ако срещнем в даден момент разделител, то в буфера би трябвало да стои поредната дума. Можем да я анализираме дали е с малки или главни букви и да зачистим буфера. Това можем да повтаряме до достигане на края на файла. Изглежда по-ефективно, нали?

За по-ефективно проверяване за главни/малки букви можем да направим цикъл по буквите и проверка на всяка буква. Така ще си спестим преобразуването в горен/долен регистър, което заделя излишно памет за всяка проверена дума, която след това се освобождава, и в крайна сметка това отнема процесорно време.

Очевидно второто решение е по-ефективно. Възниква въпросът дали трябва, след като сме написали първото решение, да го изхвърлим и да напишем съвсем друго. Всичко зависи от изискванията за ефективност. В условието на задачата няма предпоставки да смятаме, че ще ни подадат като вход стотици мегабайти. Следователно сегашното решение, макар и не оптимално, също е коректно и ще ни свърши работа.

Задача 2: Матрица с прости числа

Напишете програма, която прочита от стандартния вход цяло положително число N и отпечатва първите N^2 прости числа в квадратна матрица с размери N x N. Запълването на матрицата трябва да става по редове от първия към последния и отляво надясно.

Забележка: Едно естествено число наричаме просто, ако няма други делители освен 1 и себе си. Числото 1 не се счита за просто.

Примерен вход:

```
2                      3                      4
```

Примерен изход:

```
2 3                    2 3 5                  2 3 5 7
5 7                    7 11 13                11 13 17 19
                       17 19 23               23 29 31 37
                                              41 43 47 53
```

Намиране на подходяща идея за решение

Можем да решим задачата като с помощта на два вложени цикъла отпечатаме редовете и колоните на резултатната матрица. За всеки неин елемент ще извличаме и отпечатваме поредното просто число.

Разбиване на задачата на подзадачи

Трябва да решим поне две подзадачи – намиране на поредното просто число и отпечатване на матрицата. Отпечатването на матрицата можем да направим директно, но за намирането на поредното просто число ще трябва да помислим малко. Може би най-интуитивният начин, който ни идва наум за това, е, започвайки от предходното намерено просто число, да проверяваме всяко следващо дали е просто и в момента, в който това се окаже истина, да го върнем като резултат. Така на хоризонта се появява още една подзадача – проверка дали дадено число е просто.

Проверка на идеята

Нашата идея за решение на задачата директно получава търсения в условието резултат. Разписваме 1-2 примера на хартия и се убеждаваме, че работи.

Да помислим за структурите от данни

В тази задача се ползва една единствена структура от данни – матрицата. Естествено е да използваме двумерен масив.

Да помислим за ефективността

Тъй като изходът е на конзолата, при особено големи матрици (например 1000 x 1000) резултатът няма да може да се визуализира добре. Това означава, че задачата трябва да се реши за разумно големи матрици, но не прекалено големи, например за N ≤ 200. При нашия алгоритъм при N=200 ще трябва да намерим първите 40 000 прости числа, което не би трябвало да е бавно.

Стъпка 1 – Проверка дали дадено число е просто

За проверката дали дадено число е просто можем да дефинираме метод **`IsPrime(…)`**. За целта е достатъчно да проверим, че то не се дели без остатък на никое от предхождащите го числа. За да сме още по-точни, достатъчно е да проверим, че то не се дели на никое от числата между 2 и корен квадратен от числото. Това е така, защото, ако числото **`p`** има делител **`x`**, то **`p = x.y`** и поне едно от числата **`x`** и **`y`** ще е по-малко или равно на корен квадратен от **`p`**. Следва реализация на метода:

```
private static bool IsPrime (int number)
{
   int maxDivider = (int)Math.Sqrt(number);
   for (int divider = 2; divider <= maxDivider; divider++)
   {
      if (number % divider == 0)
      {
         return false;
      }
   }
   return true;
}
```

Сложността на горния пример е `O(Sqrt(number))`, защото правим най-много корен квадратен от number проверки. Тази сложност ще ни свърши работа в тази задача, но дали не може този метод да се оптимизира още малко? Ако се замислим, всяко второ число е четно, а всички четни числа се делят на 2. Тогава горният метод безсмислено ще проверява всички четни числа до корен квадратен от number в случай, че числото, което проверяваме, е нечетно. Как можем да премахнем тези ненужни проверки? Още в началото на метода можем да проверим дали числото се дели на 2 и после да организираме основния цикъл така, че да прескача проверката на четните делители. Новата сложност, която ще получим е O(`Sqrt(number) / 2`).

Това е пример как можем да оптимизираме вече написан метод.

```
private static bool IsPrime(int number)
{
  if (number == 2)
  {
    return true;
  }
  if (number % 2 == 0)
  {
    return false;
  }

  int maxDivider = (int)Math.Sqrt(number);
  for (int divider = 3; divider <= maxDivider; divider += 2)  {
    if (number % divider == 0)
    {
      return false;
    }
  }
  return true;
}
```

Както виждаме, кодът на метода се е изменил минимално спрямо неоптимизираната версия.

Можем да се уверим, че и двата метода работят коректно, подавайки им последователно различни числа, някои от които прости, и проверявайки върнатия резултат.

⚠	**Преди да оптимизирате даден метод трябва да го тествате, за да сте сигурни, че работи.**

Причината е, че след оптимизирането, кодът най-често става по-голям, по-труден за четене и съответно по-труден за дебъгване в случай, че не работи правилно.

⚠	**Бъдете внимателни, когато оптимизирате код. Не изпадайте в крайности и не правете ненужни оптимизации, които правят кода минимално по-бърз, но за сметка на това драстично влошават четливостта и затрудняват поддръжката на кода.**

Стъпка 2 – Намиране на следващото просто число

За намирането на следващото просто число можем да дефинираме метод, който приема като параметър дадено число, и връща като резултат първото, по-голямо от него, просто число. За проверката дали числото е просто ще използваме метода от предишната стъпка. Следва реализацията на метода:

```
private static int FindNextPrime(int startNumber)
{
   int number = startNumber;
   while(!IsPrime(number))
   {
      number++;
   }
   return number;
}
```

Отново трябва да изпробваме метода, подавайки му няколко числа и проверявайки дали резултатът е правилен.

Стъпка 3 – Отпечатване на матрицата

След като дефинирахме горните методи, вече сме готови да отпечатаме и цялата матрица:

```
private static void PrintMatrix(int dimension)
{
   int lastPrime = 1;
   for (int row = 0; row < dimension; row++)
   {
      for (int col = 0; col < dimension; col++)
      {
         int nextPrime = FindNextPrime(lastPrime + 1);
         Console.Write("{0,4}", nextPrime);
         lastPrime = nextPrime;
      }
      Console.WriteLine();
   }
}
```

Стъпка 4 – Вход от конзолата

Остава да добавим възможност за прочитане на N от конзолата:

```
static void Main()
{
   int n = ReadInput();
   PrintMatrix(n);
}

private static int ReadInput()
{
  Console.Write ("N = ");
  string input = Console.ReadLine();
  int n = int.Parse(input);
  return n;
}
```

Тестване на решението

След като всичко е готово, можем да пристъпим към проверка на решението. За целта можем да намерим например първите 25 прости числа и да проверим изхода на програмата за стойности на N от 1 до 5. Не трябва да пропускаме случая за N=1, тъй като това е граничен случай и вероятността за допусната грешка при него е значително по-голяма.

В конкретния случай, при условие че сме тествали добре методите на всяка стъпка, можем да се ограничим с примерите от условието на задачата. Ето как изглежда изходът от програмата за стойности на N съответно 1, 2, 3 и 4:

```
2        2 3               2 3 5               2 3 5 7
         5 7               7 11 13             11 13 17 19
                           17 19 23            23 29 31 37
                                               41 43 47 53
```

Можем да се уверим, че решението на задачата работи сравнително бързо и за по-големи стойности на N. Примерно при N=200 не се усеща някакво забавяне.

Следва пълната реализация на решението:

PrimesMatrix.cs

```
using System;

public class PrimesMatrix
{
  static void Main()
  {
    int n = ReadInput();
    PrintMatrix(n);
  }

  private static int ReadInput()
  {
    Console.Write("N = ");
    string input = Console.ReadLine();
    int n = int.Parse(input);
    return n;
    }

    private static bool IsPrime(int number)
    {
        if (number == 2)
        {
            return true;
        }
        if (number % 2 == 0)
        {
            return false;
        }

        int maxDivider = (int)Math.Sqrt(number);
        for (int divider = 3; divider <= maxDivider; divider += 2)
        {
            if (number % divider == 0)
            {
                return false;
            }
        }
        return true;
    }

    private static int FindNextPrime(int startNumber)
    {
        int number = startNumber;
        while(!IsPrime(number))
        {
            number++;
        }
        return number;
    }

    private static void PrintMatrix(int dimension)
    {
        int lastPrime = 1;
        for (int row = 0; row < dimension; row++)
        {
            for (int col = 0; col < dimension; col++)
            {
                int nextPrime = FindNextPrime(lastPrime + 1);
                Console.Write("{0,4}", nextPrime);
                lastPrime = nextPrime;
            }
            Console.WriteLine();
        }
    }
}
```

Дискусия за производителността

Трябва да отбележим, че посоченото решение **не търси** простите числа по най-ефективния начин. Въпреки това, с оглед яснотата на изложението и поради очаквания малък размер на матрицата, можем да използваме този алгоритъм без да имаме проблеми с производителността.

Ако трябва да подобрим производителността, можем да намерим първите N^2 числа с "решето на Ератостен" (Sieve of Eratosthenes) без да проверяваме дали всяко число е просто до намиране на N^2 прости числа.

Задача 3: Аритметичен израз

Напишете програма, която изчислява стойността на прост аритметичен израз, съставен от цели числа без знак и аритметичните операции "+" и "-". Между числата няма интервали.

Изразът се задава във формат:

```
<число><операция>...<число>
```

Примерен вход:

```
1+2-7+2-1+28+2+3-37+22
```

Примерен изход:

```
15
```

Намиране на подходяща идея за решение

За решаване на задачата можем да използваме факта, че формата на израза е стриктен и ни гарантира, че имаме последователност от число, операция, отново число и т.н.

Така можем да извлечем всички числа участващи в израза, след това всички оператори и накрая да изчислим стойността на израза, комбинирайки числата с операторите.

Проверка на идеята

Наистина, ако вземем лист и химикал и изпробваме подхода с няколко израза, получаваме верен резултат. Първоначално резултатът е равен на първото число, а на всяка следващата стъпка добавяме или изваждаме следващото число в зависимост от текущия оператор.

Структури от данни и ефективност

Задачата е прекалено проста, за да използваме сложни структури от данни. Числата и знаците можем да пазим в масив. За проблеми с ефективността не може да говорим, тъй като всеки знак и всяко число се обработват точно по веднъж, т.е. имаме линейна сложност на алгоритъма.

Разбиване на задачата на подзадачи

След като сме се убедили, че идеята работи можем да пристъпим към разбиването на задачата на подзадачи. Първата подзадача, която ще трябва да решим, е извличането на числата от израза. Втората ще е извличането на операторите. Накрая ще трябва да изчислим стойността на целия израз, използвайки числата и операторите, които сме намерили.

Стъпка 1 – Извличане на числата

За извличане на числата е необходимо да разделим израза, като за разделители използваме операторите. Това можем да направим лесно чрез метода **`Split(…)`** на класа **`String`**. След това ще трябва да преобразуваме получения масив от символни низове в масив от цели числа:

```
private static int[] ExtractNumbers(string expression)
{
   string[] splitResult = expression.Split('+', '-');

   List<int> numbers = new List<int>();
   foreach (string number in splitResult)
   {
      numbers.Add(int.Parse(number));
   }
   return numbers.ToArray();
}
```

За преобразуването на символните низове в цели числа използваме метода **`Parse(…)`** на класа **`Int32`**. Той приема като параметър символен низ и връща като резултат целочислената стойност, представена от него.

Защо използваме масив за съхранение на числата? Не можем ли да използваме например свързан списък или динамичен масив? Разбира се, че можем, но в случая е нужно единствено да съхраним числата и след това да ги обходим при изчисляването на резултата. Ето защо масивът ни е напълно достатъчен.

Преди да преминем към следващата стъпка проверяваме дали извличането на числата работи коректно:

```
static void Main()
{
    int[] numbers = ExtractNumbers("1+2-7+2-1+28");
    foreach (int x in numbers)
    {
        Console.Write("{0} ", x);
    }
}
```

Резултатът е точно такъв, какъвто трябва да бъде:

```
1 2 7 2 1 28
```

Проверяваме и граничния случай, когато изразът се състои само от едно число без оператори, и се уверяваме, че и той се обработва добре.

Стъпка 2 – Извличане на операторите

Извличането на операторите можем да направим, като последователно обходим низа и проверим всяка буквичка дали отговаря на операциите от условието:

```
private static char[] ExtractOperators(string expression)
{
    string operatorCharacters = "+-";
    List<char> operators = new List<char>();
    foreach (char c in expression)
    {
        if(operatorCharacters.Contains(c))
        {
            operators.Add(c);
        }
    }
    return operators.ToArray();
}
```

Следва проверка дали методът работи коректно:

```
static void Main()
{
    char[] operators = ExtractOperators("1+2-7+2-1+28");
    foreach(char oper in operators)
    {
        Console.Write("{0} ", oper);
    }
}
```

Изходът от изпълнението на програмата е правилен:

```
+ - + - +
```

Правим проверка и за граничния случай, когато изразът не съдържа оператори, а се състои само от едно число. В този случай получаваме празен низ, което е очакваното поведение.

Стъпка 3 – Изчисляване на стойността на израза

За изчисляване на стойността на израза можем да използваме факта, че числата винаги са с едно повече от операторите и с помощта на един цикъл да изчислим стойността на израза при условие, че са ни дадени списъците с числата и операторите:

```
private static int CalculateExpression(int[] numbers,
   char[] operators)
{
   int result = numbers[0];
   for (int i = 1; i < numbers.Length; i++)
   {
      char operation = operators[i - 1];
      int nextNumber = numbers[i];
      if (operation == '+')
      {
         result += nextNumber;
      }
      else if (operation == '-')
      {
         result -= nextNumber;
      }
   }
   return result;
}
```

Проверяваме работата на метода:

```
static void Main()
{
   // Expression: 1 + 2 - 3 + 4
   int[] numbers = new int[] { 1, 2, 3, 4 };
   char[] operators = new char[] { '+', '-', '+' };
   int result = CalculateExpression(numbers, operators);
   // Expected result is 4
   Console.WriteLine(result);
}
```

Резултатът е коректен:

```
4
```

Стъпка 4 – Вход от конзолата

Ще трябва да дадем възможност на потребителя да въвежда израз:

```
private static string ReadExpression()
{
   Console.WriteLine("Enter expression:");
   string expression = Console.ReadLine();
   return expression;
}
```

Стъпка 5 – Сглобяване на всички части в едно цяло

Остава ни само да накараме всичко да работи заедно:

```
static void Main()
{
   string expression = ReadExpression();

   int[] numbers = ExtractNumbers(expression);
   char[] operators = ExtractOperators(expression);

   int result = CalculateExpression(numbers, operators);
   Console.WriteLine("{0} = {1}", expression, result);
}
```

Тестване на решението

Можем да използваме примера от условието на задачата при тестването на решението. Получаваме коректен резултат:

```
Enter expression:
1+2-7+2-1+28+2+3-37+22
1+2-7+2-1+28+2+3-37+22 = 15
```

Трябва да направим още няколко теста с различни примери, които да включват и случая, когато изразът се състои само от едно число, за да се уверим, че решението ни работи.

Можем да тестваме и празен низ. Не е много ясно дали това е коректен вход, но можем да го предвидим за всеки случай. Освен това не е ясно какво става, ако някой въведе интервали в израза, например вместо "**2+3**" въведе "**2 + 3**". Хубаво е да предвидим тези ситуации.

Друго, което забравихме да тестваме, е какво става при число, което не се събира в типа **int**. Какво ще стане, ако ни бъде подаден изразът "**11111111111111111111111111111+222222222222222222222222222222**"?

Дребни поправки и повторно тестване

Във всички случаи, когато изразът е невалиден, ще се получи някакво изключение (най-вероятно **`System.FormatException`**). Достатъчно е да прихванем изключенията и при настъпване на изключение да съобщим, че е въведен грешен израз. Следва пълната реализация на решението след тази корекция:

SimpleExpressionEvaluator.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class SimpleExpressionEvaluator
{
   private static int[] ExtractNumbers(string expression)
   {
      string[] splitResult = expression.Split('+', '-');
      List<int> numbers = new List<int>();
      foreach (string number in splitResult)
      {
         numbers.Add(int.Parse(number));
      }
      return numbers.ToArray();
   }

   private static char[] ExtractOperators(string expression)
   {
      string operationsCharacters = "+-";
      List<char> operators = new List<char>();
      foreach (char c in expression)
      {
         if (operationsCharacters.Contains(c))
         {
            operators.Add(c);
         }
      }
      return operators.ToArray();
   }

   private static int CalculateExpression(int[] numbers,
      char[] operators)
   {
      int result = numbers[0];
      for (int i = 1; i < numbers.Length; i++)
      {
         char operation = operators[i - 1];
```

```
            int nextNumber = numbers[i];
            if (operation == '+')
            {
                result += nextNumber;
            }
            else if (operation == '-')
            {
                result -= nextNumber;
            }
        }
        return result;
    }

    private static string ReadExpression()
    {
        Console.WriteLine("Enter expression:");
        string expression = Console.ReadLine();
        return expression;
    }

    static void Main()
    {
        try
        {
            string expression = ReadExpression();

            int[] numbers = ExtractNumbers(expression);
            char[] operators = ExtractOperators(expression);

            int result = CalculateExpression(numbers, operators);
            Console.WriteLine("{0} = {1}", expression, result);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Invalid expression!");
        }
    }
}
```

Упражнения

1. Решете задачата "броене на думи в текст", използвайки само един буфер за четене (**`StringBuilder`**). Промени ли се сложността на алгоритъмът ви?
2. Реализирайте по-ефективно решение на задачата "матрица с прости числа" като търсите простите числа с "решето на Ератостен": http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes.

3. Добавете поддръжка на операциите умножение и целочислено деление в задачата "аритметичен израз". Имайте предвид, че те са с по-висок приоритет от събирането и изваждането!
4. Добавете поддръжка на реални числа, не само цели.
5. Добавете поддръжка на скоби в задачата "аритметичен израз".
6. Напишете програма, която валидира аритметичен израз. Например "2*(2.25+5.25)-17/3" е валиден израз, докато "*232*-25+(33+a" е невалиден.

Решения и упътвания

1. Можете да четете входния файл символ по символ. Ако поредният символ е буква, го добавяте към буфера, а ако е разделител, анализирате буфера (той съдържа поредната дума) и след това зачиствате буфера. Когато свърши входния файл, трябва да анализирате последната дума, която е в буфера (ако файлът не завършва с разделител).
2. Помислете първо колко прости числа ви трябват. След това помислете до каква стойност трябва да пускате "решето на Ератостен", за да ви стигнат простите числа за запълване на матрицата. Можете опитно да измислите някаква формула.
3. Достатъчно е да изпълните първо всички умножения и деления, а след тях всички събирания. Помислихте ли за деление на нула?
4. Работата с реални числа можете да осигурите като разширите използването на символа "." и заместите **`int`** с **`double`**.
5. Можем да направим следното: намираме първата затваряща скоба и търсим наляво съответната ѝ отваряща скоба. Това, което е в скобите, е аритметичен израз без скоби, за който вече имаме алгоритъм за изчисление на стойността му. Можем да го заместим със стойността му. Повтаряме това за следващите скоби докато скобите свършат. Накрая ще имаме израз без скоби.

   Например, ако имаме "2*((3+5)*(4-7*2))", ще заместим "(3+5)" с 8, след това "(4-7*2)" с -10. Накрая ще заместим (8*-10) с -80 и ще сметнем 2*-80, за да получим резултата -160. Трябва да предвидим аритметични операции с отрицателни числа, т.е. да позволяваме числата да имат знак.

   Съществува и друг алгоритъм. Използва се стек и преобразуване на израза до "обратен полски запис". Можете да потърсите в Интернет за фразата "**`postfix notation`**" и за "**`shunting yard algorithm`**".
6. Ако изчислявате израза с обратен полски запис, можете да допълните алгоритъма, така че да проверява за валидност на израза. Добавете следните правила: когато очаквате число, а се появи нещо друго, изразът е невалиден. Когато очаквате аритметична операция, а се

появи нещо друго, изразът е невалиден. Когато скобите не си съответстват, ще препълните стека или ще останете накрая с недоизпразнен стек. Помислете за специални случаи, например "`-1`", "`-(2+4)`" и др.

Глава 26. Практически задачи за изпит по програмиране – тема 3

В тази тема...

В настоящата тема ще разгледаме условията и ще предложим решения на няколко примерни за изпит. При решаването на задачите ще се придържаме към съветите от главата "Как да решаваме задачи по програмиране".

Задача 1: Квадратна матрица

По дадено число N (въвежда се от клавиатурата) да се генерира и отпечата квадратна матрица, съдържаща числата от 0 до N^2-1, разположени като спирала, започваща от центъра на матрицата и движеща се по часовниковата стрелка, тръгвайки в началото надолу (вж. примерите).

Примерен резултат при N=3 и N=4:

4	5	6
3	0	7
2	1	8

15	4	5	6
14	3	0	7
13	2	1	8
12	11	10	9

Решение на задачата

От условието лесно се вижда, че имаме поставена алгоритмична задача. Основната част от решението на задачата – да измислим подходящ алгоритъм за запълване на клетките на квадратна матрица по описания начин. Ще покажем на читателя типичните разсъждения необходими за решаването на този конкретен проблем.

Да започнем с избора на структура от данни за представяне на матрицата. Удобно е да имаме директен достъп до всеки елемент на матрицата, затова ще се спрем на двумерен масив **`matrix`** от целочислен тип. При стартирането на програмата прочитаме от стандартния вход размерността **`n`** на матрицата и я инициализираме по следния начин:

```
int[,] matrix = new int[n,n];
```

Измисляне на идея за решение

Следващата стъпка е да измислим идеята на алгоритъма, който ще имплементираме. Трябва да запълним матрицата с числата от 0 до N^2-1 и веднага съобразяваме, че това може да стане с помощта на цикъл, който на всяка итерация поставя едно от числата в предназначената за него клетка на матрицата. Текущата позиция ще представяме чрез целочислените променливи **`positionX`** и **`positionY`** – двете координати на позицията. Да приемем, че знаем началната позиция – тази, на която трябва да поставим първото число. По този начин задачата се свежда до намиране на метод за определяне на всяка следваща позиция, на която трябва да бъде поставено число – това е нашата главна подзадача.

Подходът за определяне на следващата позиция спрямо текущата е следният: търсим строга закономерност на промяната на индексите при спираловидното движение по клетките. Започваме от най-очевидното нещо – движението винаги е по посока на часовниковата стрелка, като първоначално посоката е надолу. Дефинираме целочислена променлива **`direction`**, която ще показва текущата посока на движение. Тази променлива ще приема стойностите 0 (надолу), 1 (наляво), 2 (нагоре) и 3 (надясно). При смяна на посоката на движение просто увеличаваме с единица стойността на direction и делим по модул 4 (за да получаваме само стойности от 0 до 3).

Следващата стъпка при съставянето на алгоритъма е да установим кога се сменя посоката на движение (през колко итерации на цикъла). От двата примера можем да забележим, че броят на итерациите, през които се сменя посоката образува нестрого растящите редици 1, 1, 2, 2, 2 и 1, 1, 2, 2, 3, 3, 3. Ако разпишем на лист хартия по-голяма матрица от същия вид ясно виждаме, че редицата на смените на посоката следва същата схема – числата през едно нарастват с 1, като последното число не нараства. За моделирането на това поведение ще използваме променливите **`stepsCount`** (броят на итерациите в текущата посока), **`stepPosition`** (номерът на поредната итерация в тази посока) и **`stepChange`** (флаг, показващ дали на текущата итерация трябва да увеличим стойността на **`stepCount`**).

Проверка на идеята

Нека проверим идеята. След директно разписване на алгоритъма за N равно на 0, 1, 2 и 3 се вижда, че той е коректен и можем да преминем към неговата реализация.

Структури от данни и ефективност

При тази задачата избора на структурите от данни е еднозначен. Матрицата ще пазим в двумерен масив. Други данни нямаме (освен числа). С ефективността няма да имаме проблем, тъй като програмата ще направи толкова стъпки, колкото са елементите в матрицата, т.е. имаме линейна сложност.

Реализация на идеята: стъпка по стъпка

Нека видим как можем да реализираме тази идея като код:

```
for (int i = 0; i < count; i++)
{
  matrix[positionY, positionX] = i;
  if (stepPosition < stepsCount)
  {
    stepPosition++;
  }
  else
  {
        stepPosition = 1;
        if (stepChange == 1)
        {
            stepsCount++;
        }
        stepChange = (stepChange + 1) % 2;
        direction = (direction + 1) % 4;
    }

    switch (direction)
    {
        case 0:
            positionY++;
            break;
        case 1:
            positionX--;
            break;
        case 2:
            positionY--;
            break;
        case 3:
            positionX++;
            break;
    }
}
```

Тук е моментът да отбележим, че е голяма рядкост да съставим тялото на подобен цикъл от първия път, без да сгрешим. Вече знаем за правилото да пишем кода стъпка по стъпка, но за тялото на този цикъл то е трудно приложимо – нямаме ясно обособени подзадачи, които можем да тестваме независимо една от друга. Това не бива да ни притеснява – можем да използваме мощния debugger на Visual Studio за постъпково проследяване на изпълнението на кода. По този начин лесно ще открием къде е грешката, ако има такава.

След като имаме добре измислена идея на алгоритъм (дори да не сме напълно сигурни, че така написаният код работи безпроблемно), остава да дадем начални стойности на вече дефинираните променливи и да отпечатаме получената след изпълнението на цикъла матрица.

Ясно е, че броят на итерациите на цикъла е точно N^2 и затова инициализираме променливата `count` с тази стойност. От двата дадени примера и нашите собствени (написани на лист) примери определяме началната позиция в матрицата в зависимост от четността на нейната размерност:

```
int positionX = n / 2;
int positionY = n % 2 == 0 ? ((n / 2) - 1 : (n / 2));
```

На останалите променливи даваме еднозначно следните стойности (вече обяснихме каква е тяхната семантика):

```
int direction = 0;
int stepsCount = 1;
int stepPosition = 0;
int stepChange = 0;
```

Последната подзадача, която трябва да решим, за да имаме работеща програма, е отпечатването на матрицата на стандартния изход. Това става най-лесно с два вложени цикъла, които я обхождат по редове и на всяка итерация на вътрешния цикъл прилагаме подходящото форматиране:

```
for (int i = 0; i < n; i++)
{
   for (int j = 0; j < n; j++)
   {
      Console.Write("{0,3}", matrix[i, j]);
   }
   Console.WriteLine();
}
```

С това изчерпахме основните съставни елементи на програмата. Следва пълният изходен код на нашето решение:

MatrixSpiral.cs

```
public class MatrixSpiral
{
   static void Main()
   {
      Console.Write("N = ");
      int n = int.Parse(Console.ReadLine());
      int[,] matrix = new int[n, n];

      FillMatrix(matrix, n);

      PrintMatrix(matrix, n);
   }

   private static void FillMatrix(int[,] matrix, int n)
   {
      int count = n * n;
      int positionX = n / 2;
      int positionY = n % 2 == 0 ? ((n / 2) - 1 : (n / 2));

      int direction = 0;
      int stepsCount = 1;
      int stepPosition = 0;
```

```
    int stepChange = 0;

    for (int i = 0; i < count; i++)
    {
      matrix[positionY, positionX] = i;
      if (stepPosition < stepsCount)
      {
        stepPosition++;
      }
      else
      {
        stepPosition = 1;

        if (stepChange == 1)
        {
          stepsCount++;
        }
        stepChange = (stepChange + 1) % 2;
        direction = (direction + 1) % 4;
      }

      switch (direction)
      {
        case 0:
          positionY++;
          break;
        case 1:
          positionX--;
          break;
        case 2:
          positionY--;
          break;
        case 3:
          positionX++;
          break;
      }
    }
  }

  private static void PrintMatrix(int[,] matrix, int n)
  {
    for (int i = 0; i < n; i++)
    {
      for (int j = 0; j < n; j++)
      {
        Console.Write("{0,3}", matrix[i, j]);
      }
      Console.WriteLine();
    }
    }
}
```

Тестване на решението

След като сме имплементирали решението, уместно е да го тестваме с достатъчен брой стойности на N, за да се уверим, че работи правилно. Започваме с примерните стойности 3 и 4, а после проверяваме и за 5, 6, 7, 8, 9, … Важно е да тестваме и за граничните случаи: 0 и 1. Провеждаме необходимите тестове и се убеждаваме, че всичко работи. В случая не е уместно да тестваме за скорост (примерно с N=1000), защото при голямо N изходът е прекалено обемен и задачата няма особен смисъл.

Задача 2: Броене на думи в текстов файл

Даден е текстов файл `words.txt`, който съдържа няколко думи, по една на ред. Да се напише програма, която намира броя срещания на всяка от дадените думи като подниз във файла `sample.txt`. Главните и малките букви се считат за еднакви. Резултатът да се запише в текстов файл с име `result.txt` във формат **`<дума> - <брой срещания>`**.

Примерен входен файл `words.txt`:

```
For
academy
student
develop
```

Примерен входен файл `sample.txt`:

```
The Telerik Academy for .NET software development engineers is a
famous center for professional training of .NET experts. Telerik
Academy offers courses designed to develop practical computer
programming skills. Students graduated the Academy are guaranteed
to have a job as a software developers in Telerik.
```

Примерен резултатен файл `result.txt`:

```
for – 2
academy – 3
student – 1
develop – 3
```

Решение на задачата

В дадената задача акцентът е не толкова върху алгоритъма за нейното решаването, а по-скоро върху техническата реализация. За да напишем решението, трябва да сме добре запознати с работата с файлове в C#, както и с основните структури от данни.

Измисляне на идея за решение

При тази задача идеята за решение е очевидна: прочитаме файла с думите, след това минаваме през текста и за всяка дума в него проверяваме дали е от интересните за нас думи и ако е увеличаваме съответния брояч. Измисляме решението бързо, защото от алгоритмична гледна точка е лесно и интуитивно.

Проверка на идеята

Идеята за решаване е тривиална, но все пак можем да я проверим като разпишем на лист хартия какво ще се получи за примерния входен файл. Лесно се убеждаваме, че тази идея е правилна.

Разделяме задачата на подзадачи

При реализацията на програмата можем да отделим три основни стъпки (подзадачи):

1. Прочитаме файла **`words.txt`** и добавяме всяка дума от него към списък **`words`** (за целта използваме **`List<string>`** в реализацията). За четенето на текстови файлове е удобно да използваме методи на класа **`File`**, който вече сме разгледали в предходните глави.
2. Обхождаме в цикъл всяка дума от файла **`sample.txt`** и проверяваме дали тя съвпада с някоя дума от списъка **`words`**. За четенето на думите от файла отново използваме класа **`File`**. При проверката игнорираме разликата между малки и големи букви. В случай на съвпадение с вече добавена дума увеличаваме броя на срещанията на съответната дума от списъка **`words`**. Броят на срещанията на думите съхраняваме в целочислен масив **`wordsCount`**, в който елементите съвпадат позиционно с елементите на списъка **`words`**.
3. Записваме резултата от така извършеното преброяване във файла **`result.txt`**, спазвайки формата, зададен в условието. За отваряне и писане във файла е удобно да използваме отново класа **`File`**.

Имплементация

Директно следваме стъпките, които идентифицирахме и ги реализираме. Получаваме следния сорс код:

WordsCounter.cs

```
using System.Collections.Generic;
using System.IO;

public class WordsCounter
{
  static void Main()
  {
    List<string> words = new List<string>();
    foreach(string word in File.ReadAllLines("words.txt"))
    {
      words.Add(word.ToLower());
    }

    int[] wordsCount = new int[words.Count];

    string[] sampleFileWords =
      File.ReadAllText("sample.txt").Split(' ', '.');

    foreach(string sampleWordRaw in sampleFileWords)
    {
      string sampleWord = sampleWordRaw.ToLower();
      foreach (string word in words)
      {
        if (sampleWord.Contains(word))
        {
          wordsCount[words.IndexOf(word)]++;
        }
      }
    }

    using(StreamWriter resultFile  =
      File.CreateText("result.txt"))
    {
      foreach (string word in words)
      {
        resultFile.WriteLine("{0} - {1}", word,
          wordsCount[words.IndexOf(word)]);
      }
    }
  }
}
```

Ефективност на решението

Май подценихме задачата и избързахме да напишем сорс кода. Ако се върнем към препоръките от главата: "Как да решаваме задачи по програмиране", ще видим, че пропуснахме една важна стъпка: **избор на подходящи структури от данни**. Написахме кода като използвахме първата възможна структура от данни, за която се сетихме, но не помислихме дали има по-добър вариант.

Време е да вмъкнем няколко думи за бързодействието (ефективността) на нашето решение. В повечето случаи така написаната програма ще работи достатъчно бързо за голям набор от входни данни, което я прави приемливо решение при явяване на изпит. Въпреки това, е възможно да възникне ситуация, в която файлът **`words.txt`** съдържа много голям брой думи (примерно 10 000), което ще доведе до голям брой елементи на списъка **`words`**. Причината да се интересуваме от това е методът **`indexOf(…)`**, който използваме за намиране на индекса на дадена дума. Неговото бързодействие е обратно пропорционално на броя на елементите на списъка и в този случай ще имаме осезаемо забавяне при работата на програмата. Например при 10 000 думи търсенето на една дума ще изисква 10 000 сравнения на двойки думи. Това ще се извърши толкова пъти, колкото са думите в текста, а те може да са много, да кажем 200 000. Тогава решението ще работи осезаемо бавно.

Можем да решим описания проблем като използваме хеш-таблица вместо целочисления масив **`wordsCount`** в горния код. Ще пазим в хеш-таблицата като ключове всички думи, които срещаме в текста, а като стойности ще пазим колко пъти се среща съответната дума. По този начин няма да се налага последователно търсене в списъка **`words`**, защото хеш-таблицата имплементира значително по-бързо асоциативно търсене сред своите елементи. Можеше да се сетим за това, ако бяхме помислили за структурите от данни преди да се хвърлим да пишем сорс кода. Май трябваше да се доверим на методологията за решаване на задачи, а не да действаме както си знаем, нали?

Нека видим подобрения по този начин вариант на решението:

WordsCounter.cs

```
using System.Collections.Generic;
using System.IO;

public class WordsCounter
{
   static void Main()
   {
      List<string> words = new List<string>();
      foreach(string word in File.ReadAllLines("words.txt"))
      {
         words.Add(word.ToLower());
      }

      Dictionary<string, int> wordsCount =
         new Dictionary<string, int>();

      string[] sampleFileWords =
         File.ReadAllText("sample.txt").Split(' ', '.');
```

```
    foreach(string sampleWordRaw in sampleFileWords)
    {
      string sampleWord = sampleWordRaw.ToLower();

      foreach (string word in words)
      {
        if (sampleWord.Contains(word))
        {
          if (wordsCount.ContainsKey(word))
          {
            wordsCount[word] = wordsCount[word] + 1;
          }
          else
          {
            wordsCount[word] = 1;
          }
        }
      }
    }

    using(StreamWriter resultFile  =
      File.CreateText("result.txt"))
    {
      foreach (string word in words)
      {
        int count = wordsCount.ContainsKey(word) ?
          wordsCount[word] : 0;
        resultFile.WriteLine("{0} - {1}", word, count);
      }
    }
  }
}
```

Тестване на решението

Разбира се, както при всяка друга задача, е много важно да тестваме решението, което сме написали и е препоръчително да измислим свои собствени примери освен този, който е даден в условието, и да се убедим, че изходът е коректен.

Трябва да тестваме и граничните случаи: какво става, ако единият от входните файлове е празен или и двата са празни? Какво става, ако в двата файла има само по една дума? Трябва да проверим дали малки и главни букви се считат за еднакви.

Накрая трябва да тестваме за скорост. За целта с малко copy/paste правим списък от 10 000 думи във файла **`words.txt`** и копираме текста от файла **`sample.txt`** достатъчно на брой пъти, за да достигне до 5-10 MB. Стартираме и се убеждаваме, че имаме проблем. Чакаме минута-две, но програмата не завършва. Нещо не е наред.

Търсене на проблема с бързодействието

Ако пуснем програмата през дебъгера, ще се забележим, че имаме много глупава грешка в следния фрагмент код:

```
foreach(string sampleWordRaw in sampleFileWords)
{
   string sampleWord = sampleWordRaw.ToLower();
   foreach (string word in words)
   {
      if (sampleWord.Contains(word))
      {
         if (wordsCount.ContainsKey(word))
         {
            wordsCount[word] = wordsCount[word] + 1;
         }
         else
         {
            wordsCount[word] = 1;
         }
      }
   }
}
```

Вижда се, че ако имаме 10 000 думи в масива **`words`** и 100 000 думи, които прочитаме една по една, за всяка от тях ще обходим във **`for`**-цикъл нашия масив и това прави 10 000 * 100 000 операции, които отнемат доста време. Как да оправим проблема?

Оправяне на проблема с бързодействието

За да работи коректно програмата очевидно трябва да преминем поне през веднъж през целия текст. Ако не прегледаме целия текст има опасност да не преброим някоя от думите. Следователно трябва да търсим ускорение на кода, който обработва всяка от думите. В текущата имплементация се върти цикъл до броя думи, които броим и ако те са много, този цикъл забавя чувствително програмата.

Идва ни идеята да заменим цикъла по думите, които броим с нещо по-бързо. Дали е възможно? Да помислим защо въртим този цикъл. Въртим го, за да видим дали думата, която сме прочели от текста е сред нашия списък от думи, за които броим колко пъти се срещат. Реално ни трябва бързо търсене в множество от думи. За целта може да се ползва **`HashSet`** или **`Dictionary`**, нали? Да си припомним структурите от данни множество и хеш-таблица. При тях може да се реализира изключително бързо търсене дори ако елементите са огромен брой.

Изводът е, че до момента сгрешихме на няколко пъти от прибързване. Ако бяхме помислили за структурите от данни и за ефективността преди да напишем кода, щяхме да си спестим много време и писане. Нека сега поправим грешката. Хрумва ни следната идея:

1. Правим си хеш-таблица и в нея записваме като ключове всички думи от файла **`words.txt`**. Като стойност в тези ключове записваме числото 0. Това е броят срещания на всяка дума в текста в началния момент, преди да сме започнали да го сканираме.
2. Сканираме текста дума по дума и търсим всяка от тях в хеш-таблицата. Това е бърза операция (търсене в хеш-таблица по ключ). Ако намерим думата, увеличаваме с 1 стойността в съответния ключ. Така си осигуряваме, че всяко срещане се отбелязва и накрая за всяка дума ще получим броя на срещанията ѝ.
3. Накрая сканираме думите от файла **`words.txt`** и за всяка търсим в хеш-таблицата колко пъти се среща в текста и записваме резултата в изходния файл.

С новия алгоритъм при обработката на всяка дума от текста имаме по едно търсене в хеш-таблица и нямаме претърсване на масив, което е много бавна операция. Ето как изглежда новия алгоритъм:

FastWordsCounter.cs

```
using System.Collections.Generic;
using System.IO;

public class WordsCounter
{
  static void Main()
  {
    List<string> words = new List<string>();
    Dictionary<string, int> wordsCount =
      new Dictionary<string, int>();

    foreach (string wordRaw in File.ReadAllLines("words.txt"))
    {
      string word = wordRaw.ToLower();
      words.Add(word);
      wordsCount[word] = 0;
    }

    string[] sampleFileWords =
      File.ReadAllText("sample.txt").Split(' ', '.');
    foreach(string wordRaw in sampleFileWords)
    {
      string word = wordRaw.ToLower();
      int count;
      if (wordsCount.TryGetValue(word, out count))
      {
        wordsCount[word] = count + 1;
      }
    }

    using(StreamWriter resultFile  =
      File.CreateText("result.txt"))
    {
      foreach (string word in words)
      {
        int count = wordsCount[word];
        resultFile.WriteLine("{0} - {1}", word, count);
      }
    }
  }
}
```

Повторно тестване на проблема с бързодействието

Остава да тестваме новия алгоритъм: дали е коректен и дали работи бързо. Дали е коректен лесно можем да проверим с примерите, с които сме тествали и преди. Дали работи бързо можем да тестваме с големия пример (10 000 думи и 10 MB текст). Бързо се убеждаваме, че този път дори при големи обеми текстове програмата работи бързо. Дори пускаме 20 000 думи и 100 MB файл, за да видим дали ще работи. Уверяваме се, че дори и при такъв обем данни програмата работи стабилно и с приемлива скорост (20-30 секунди на компютър от 2008 г.).

Задача 3: Училище

В едно училище учат ученици, които са разделени в учебни групи. На всяка група преподава един учител.

За учениците се пази следната информация: име и фамилия.

За всяка група се пази следната информация: наименование и списък на учениците.

За всеки учител се пази следната информация: име, фамилия и списък от групите, на които преподава. Един учител може да преподава на повече от една група.

За училището се пази следната информация: наименование, списък на учителите, списък на групите, списък на учениците.

1. Да се проектира съвкупност от класове с връзки между тях, които моделират училището.

2. Да се реализират методи за добавяне на учител, за добавяне на група и за добавяне на ученик. Списъците могат да се представят чрез масиви или чрез списъчни структури.
3. Да се реализира метод за отпечатване на информация за даден учител: име, фамилия, списък на групите, на които преподава, и списък на учениците от всяка от тези групи.
4. Да се напише примерна тестова програма, която демонстрира работата на реализираните класове и методи.

Пример:

```
Училище "Свобода". Учители: Димитър Георгиев, Христина Николова.
Група "английски език": Иван Петров, Васил Тодоров, Елена
Михайлова, Радослав Георгиев, Милена Стефанова, учител Христина
Николова.
Група "френски език": Петър Петров, Васил Василев, учител
Христина Николова.
Група "информатика": Милка Колева, Пенчо Тошев, Ива Борисова,
Милена Иванова, Христо Тодоров, учител Димитър Георгиев.
```

Решение на задачата

Това е добър пример за задача, чиято цел е да тества умението на кандидатите, явяващи се на изпита да използват ООП за моделиране на задачи от реалния свят. Ще моделираме предметната област като дефинираме взаимно свързаните класове **`Student`**, **`Group`**, **`Teacher`** и **`School`**. За да бъде изцяло изпълнено условието на задачата ще имаме нужда и от клас **`SchoolTest`**, който демонстрира работата на дефинираните от нас класове и методи.

Измисляне на идея за решение

В тази задача няма нищо за измисляне. Тя не е алгоритмична и в нея няма какво толкова да мислим. Трябва за всеки обект от описаните в условието на задачата (студенти, учители, ученици, училище и т.н.) да дефинираме по един клас и след това в този клас да дефинираме свойства, които го описват и действия, които той може да направи. Това е всичко.

Разделяме задачата на подзадачи

Имплементацията на всеки един от класовете можем да разглеждаме като подзадача на дадената:

- Клас за студентите – **`Student`**
- Клас за групите – **`Group`**
- Клас за учителите – **`Teacher`**

- Клас за училището – **School**
- Клас за тестване на останалите класове с примерни данни – **SchoolTest**

Имплементиране: стъпка по стъпка

Удачно е да започнем реализацията с класа **Student**, тъй като от условието на задачата лесно се вижда, че той не зависи от останалите три.

Класът Student

В дефиницията имаме само две полета, представляващи име и фамилия на ученика и свойството **Name**, което връща низ с името на ученика. Дефинираме го по следния начин:

Student.cs

```
public class Student
{
   private string firstName;
   private string lastName;

   public Student(string firstName, string lastName)
   {
      this.firstName = firstName;
      this.lastName = lastName;
   }

   public string Name
   {
      get
      {
         return this.firstName + " " + this.lastName;
      }
   }
}
```

Класът Group

Следващият клас, който дефинираме е **Group**. Избираме него, защото в дефиницията му се налага да използваме единствено класа **Student**. Полетата, които ще дефинираме представляват име на групата и списък с ученици, които посещават групата. За реализацията на списъка с ученици ще използваме класа **List<Student>**. Класът ще има свойствата **Name** и **Students**, които извличат стойностите на двете полета. Добавяме два метода, които ни трябват – **AddStudent(…)** и **PrintStudents(…)**. Методът **AddStudent(…)** добавя обект от тип **Student** към списъка **students**, а методът **`PrintStudents(…)`** отпечатва името на групата и имената на учениците в нея. Нека сега видим цялата реализация на класа:

`Group.cs`

```
using System.Collections.Generic;
using System.IO;

public class Group
{
   private string name;
   private List<Student> students;

   public Group(string name)
   {
      this.name = name;
      this.students = new List<Student>();
   }

   public string Name
   {
      get
      {
         return this.name;
      }
   }

   public IEnumerable<Student> Students
   {
      get
      {
         return this.students;
      }
   }

   public void AddStudent(Student student)
   {
      students.Add(student);
   }

   public void PrintStudents(TextWriter output)
   {
      output.WriteLine("Group name: {0}", this.Name);
      output.WriteLine("Students in group:");
      foreach (Student student in this.Students)
      {
         output.WriteLine("Name: {0}", student.Name);
      }
   }
}
```

Класът Teacher

Нека сега дефинираме класа **`Teacher`**, който използва класа **`Group`**. Неговите полета са име, фамилия и списък с групи. Той има методи **`AddGroup(…)`** и **`PrintGroups(…)`**, аналогични на тези в класа **`Group`**. Методът **`PrintGroups(…)`** отпечатва името на учителя и извиква метода **`PrintStudents(…)`** на всяка група от списъка с групи:

Teacher.cs

```
using System.Collections.Generic;
using System.IO;

public class Teacher
{
  private string firstName;
  private string lastName;
  private List<Group> groups;

  public Teacher(string firstName, string lastName)
  {
    this.firstName = firstName;
    this.lastName = lastName;
    this.groups = new List<Group>();
  }

  public void AddGroup(Group group)
  {
    this.groups.Add(group);
  }

  public void PrintGroups(TextWriter output)
  {
    output.WriteLine("Teacher name: {0} {1}", this.firstName,
      this.lastName);
    output.WriteLine("Groups of teacher:");
    foreach (Group group in this.groups)
    {
      group.PrintStudents(output);
    }
  }
}
```

Класът School

Завършваме обектния модел с дефиницията на класа **`School`**, който използва всички вече дефинирани класове. Полетата му са име, списък с учители, списък с групи и списък с ученици. Пропъртитата **`Name`** и **`Teachers`** използваме за извличане на нужните данни. Дефинираме методи **`AddTeacher(…)`** и **`AddGroup(…)`** за добавяне на съответните обекти. За удобство при създаването на обектите, в метода **`AddGroup(…)`** имплементираме следната функционалност: освен добавянето на самата група като обект, добавяме към списъка с ученици и учениците, които попадат в тази група (но все още не са добавени в списъка на училището). Ето и целия код на класа:

School.cs

```
using System.Collections.Generic;

public class School
{
   private string name;
   private List<Teacher> teachers;
   private List<Group> groups;
   private List<Student> students;

   public School(string name) {
      this.name = name;
      this.teachers = new List<Teacher>();
      this.groups = new List<Group>();
      this.students = new List<Student>();
   }

   public string Name
   {
      return name;
   }

   public IEnumerable<Teacher> Teachers
   {
      get
      {
         return this.teachers;
      }
   }

   public void AddTeacher(Teacher teacher)
   {
      teachers.Add(teacher);
   }

   public void AddGroup(Group group)
   {
      groups.Add(group);
      foreach (Student student in group.Students)
      {
         if(!this.students.Contains(student))
         {
            this.students.Add(student);
         }
      }
   }
}
```

Класът TestSchool

Следва реализацията на класа **SchoolTest**, който има за цел да демонстрира класовете и методите, които дефинирахме. Това е и нашата последна подзадача – с нея решението е завършено. За демонстрацията използваме данните от примера в условието:

SchoolTest.cs

```
using System;

public class SchoolTest
{
   public static void AddObjectsToSchool(School school)
   {
      Teacher teacherGeorgiev = new Teacher("Димитър",
         "Георгиев");
      Teacher teacherNikolova = new Teacher("Христина",
         "Николова");

      school.AddTeacher(teacherGeorgiev);
      school.AddTeacher(teacherNikolova);

      // Add the English group
      Group groupEnglish = new Group("английски език");
      groupEnglish.AddStudent(new Student("Иван", "Петров"));
      groupEnglish.AddStudent(new Student("Васил", "Тодоров"));
      groupEnglish.AddStudent(new Student("Елена", "Михайлова"));
      groupEnglish.AddStudent(new Student("Радослав",
         "Георгиев"));
      groupEnglish.AddStudent(new Student("Милена",
         "Стефанова"));
      groupEnglish.AddStudent(new Student("Иван", "Петров"));

      school.AddGroup(groupEnglish);
      teacherNikolova.AddGroup(groupEnglish);

      // Add the French group
      Group groupFrench = new Group("френски език");
      groupFrench.AddStudent(new Student("Петър", "Петров"));
```

```
        groupFrench.AddStudent(new Student("Васил", "Василев"));

        school.AddGroup(groupFrench);
        teacherNikolova.AddGroup(groupFrench);

        // Add the Informatics group
        Group groupInformatics = new Group("информатика");
        groupInformatics.AddStudent(new Student("Милка",
            "Колева"));
        groupInformatics.AddStudent(new Student("Пенчо", "Тошев"));
        groupInformatics.AddStudent(new Student("Ива",
            "Борисова"));
        groupInformatics.AddStudent(new Student("Милена",
            "Иванова"));
        groupInformatics.AddStudent(new Student("Христо",
            "Тодоров"));

        school.AddGroup(groupInformatics);
        teacherGeorgiev.AddGroup(groupInformatics);
    }

    public static void Main() {
        School school = new School("Свобода");

        AddObjectsToSchool(school);

        foreach(Teacher teacher in school.Teachers)
        {
            teacher.PrintGroups(Console.Out);
            Console.WriteLine();
        }
    }
}
```

Изпълняваме програмата и получаваме очаквания резултат:

```
Teacher name: Димитър Георгиев
Groups of teacher:
Group name: информатика
Students in group:
  Name: Милка Колева
  Name: Пенчо Тошев
  Name: Ива Борисова
  Name: Милена Иванова
  Name: Христо Тодоров

Teacher name: Христина Николова
Groups of teacher:
Group name: английски език
```

```
Students in group:
  Name: Иван Петров
  Name: Васил Тодоров
  Name: Елена Михайлова
  Name: Радослав Георгиев
  Name: Милена Стефанова
  Name: Иван Петров
Group name: френски език
Students in group:
  Name: Петър Петров
  Name: Васил Василев
```

Разбира се, в реалния живот програмите не тръгват от пръв път, но в тази задача грешките, които можете да допуснете, са тривиални и няма смисъл да ги дискутираме. Всичко е въпрос на написване (ако познавате работата с класове и обектно-ориентираното програмиране като цяло).

Тестване на решението

Остава, както при всяка задача, да тестваме дали решението работи правилно. Ние вече го направихме. Може да направим и няколко теста с гранични данни, примерно група без студенти, празно училище и т.н. тестове за бързодействие няма да правим, защото задачата има неизчислителен характер. Това би било достатъчно, за да се уверим, че решението ни е коректно.

Упражнения

1. Напишете програма, която отпечатва спирална квадратна матрица, започвайки от числото 1 в горния десен ъгъл и движейки се по часовниковата стрелка. Примери при N=3 и N=4:

7	8	1
6	9	2
5	4	3

10	11	12	1
9	16	13	2
8	15	14	3
7	6	5	4

2. Напишете програма, която брои думите в текстов файл, но за дума счита всяка последователност от символи (подниз), а не само отделените с разделители. Например в текста "Аз съм студент в София" поднизовете "с", "сту", "а" и "аз съм" се срещат съответно 3, 1, 2 и 1 пъти.

3. Моделирайте със средствата на ООП файловата система в един компютър. В нея имаме устройства, директории и файлове. Устройствата са примерно твърд диск, флопи диск, CD-ROM устройство и др. Те имат име и дърво на директориите и файловете. Една директория

има име, дата на последна промяна и списък от файлове и директории, които се съдържат в нея. Един файл има име, дата на създаване, дата на последна промяна и съдържание. Файлът се намира в някоя от директориите. Файлът може да е текстов или бинарен. Текстовите файлове имат за съдържание текст (**`string`**), а бинарните – поредица от байтове (**`byte[]`**). Направете клас, който тества другите класове и показва, че с тях можем да построим модел на устройствата, директориите и файловете в компютъра.

4. Използвайки класовете от предходната задача с търсене в Интернет напишете програма, която взима истинските файлове от компютъра и ги записва във вашите класове (без съдържанието на файловете, защото няма да стигне паметта).

Решения и упътвания

1. Задачата е аналогична на първата задача от примерния изпит. Можете да модифицирате примерното решение, дадено по-горе.

2. Трябва да четете текста буква по буква и след всяка следваща буква да я долепяте към текущ буфер **`buf`** и да проверявате всяка от търсените думи за съвпадение с **`EndsWith()`**. Разбира се, няма да можете да ползвате ефективно хеш-таблица и ще имате цикъл по думите за всяка буква от текста, което не е най-бързото решение.

   Реализирането на бързо решение изисква използването на сложна структура от данни, наречена **суфиксно дърво**. Можете да потърсите в Google следното: **`"suffix tree" "pattern matching" filetype:ppt`**.

3. Задачата е аналогична на задачата с училището от примерния изпит и се решава чрез същия подход. Дефинирайте класове **`Device`**, **`Directory`**, **`File`**, **`ComputerStorage`** и **`ComputerStorageTest`**. Помислете какви свойства има всеки от тези класове и какви са отношенията между класовете. Когато тествате слагайте примерно съдържание за файловете (примерно по 1 думичка), а не оригиналното, защото то е много обемно. Помислете може ли един файл да е в няколко директории едновременно.

4. Използвайте класа **`System.IO.Directory`** и неговите статични методи **`GetFiles()`**, **`GetDirectories()`** и **`GetLogicalDrives()`**.

Telerik Academy

SCHOOL

Стани част от Академията на Телерик за ученици!

“Основната цел на обучението е да израстнеш, а нашите умове, за разлика от телата ни, могат да растат през целия ни живот!“ Мортимър Адлър

АКАДЕМИЯТА НА ТЕЛЕРИК за ученици предоставя съвременно, специализирано обучение по софтуерно инженерство за ученици от средните училища в България. Нашата основна цел е да помогнем на младите момчета и момичета да се подготвят за Националната олимпиада по информационни технологии (НОИТ) и да придобият солидни теоретични и практически умения за разработка на софтуер и софтуерни технологии.

В Училищната академия на Телерик по софтуерно инженерство:

- **Стимулираме креативността и мисленето на учениците**
- **Всеки ученик получава индивидуално внимание и съдействие**
- **Полагаме основите за писане на качествен програмен код**
- **Изучаваме съвременните софтуерни технологии**
- **Подготвяме учениците за Националната олимпиада по информационни технологии**
- **Предлагаме голямо разнообразие от практически и теоретични обучения**

Програмата на Академията включва:

C#, .NET технологии, бази данни, SQL, Silverlight, 3D графики и моделиране, разработка на игри, програмиране на микроконтролери, WPF, ASP.NET, HTML5, разработка на мобилни приложения, основи на софтуерното инженерство, работа в екип

schoolacademy.telerik.com

groups.google.com/group/it-olymp *facebook.com/TelerikSchoolAcademy*

telerik

deliver more than expected

Заключение

Ако сте стигнали до заключението и сте прочели внимателно цялата книга, приемете нашите заслужени поздравления! Убедени сме, че сте научили ценни знания за принципите на програмирането, които ще ви останат за цял живот. Дори да минат години, дори технологиите да се променят и компютрите да не бъдат това, което са в момента, фундаменталните знания за структурите от данни в програмирането и алгоритмичното мислене, както и натрупаният опит при решаването на задачи по програмиране винаги ще ви помагат, ако работите в областта на информационните технологии.

Решихте ли всички задачи?

Ако освен, че сте прочели внимателно цялата книга, сте решили и всички задачи от упражненията към всяка от главите, вие можете гордо да се наречете програмист. Всяка технология, с която ще се захванете от сега нататък, ще ви се стори лесна като детска игра. След като сте усвоили основите и фундаменталните принципи на програмирането, със завидна лекота ще се научите да ползвате бази данни и SQL, да разработвате уеб приложения и сървърен софтуер (например с ASP.NET и WCF), да пишете HTML5 приложения, да програмиране за мобилни устройства и каквото още поискате. Вие имате огромно предимство пред мнозинството от практикуващите програмиране, които не знаят какво е хеш-таблица, как работи търсенето в дървовидна структура и какво е сложност на алгоритъм. Ако наистина сте се блъскали да решите всички задачи от цялата книга, със сигурност сте постигнали едно завидно ниво на фундаментално разбиране на концепциите на програмирането и правилното мислене на програмист, което ще ви помага години наред.

Имате ли трудности със задачите?

Ако не сте решили всичките задачи от упражненията или поне голямата част от тях, върнете се и ги решете! Да, отнема много време, но това е начинът да се научите да програмирате – чрез много труд и усилия. Без да практикувате много сериозно програмирането, няма да го научите!

Ако имате затруднения, използвайте дискусионната група за курсовете по основи на програмирането, които се водят по настоящата книга в Академията на Телерик: http://groups.google.com/group/telerikacademy. Пред тези курсове са преминали няколко стотин души и голяма част от тях са решили всички задачи и са споделили решенията си, така че ги разгледайте и пробвайте, след което се опитайте да си напишете сами задачите без да гледате от тях.

На сайта на книгата (http://www.introprogramming.info) са публикувани лекции и видеообучения по настоящата книга, които могат да са много полезни, особено, ако сега навлизате за първи път в програмирането. Струва си да ги прегледате. Прегледайте също и безплатните курсовете от Академията на Телерик (http://academy.telerik.com). На техните сайтове са публикувани за свободно изтегляне всички учебни материали и видеозаписи на повечето лекции за свободно гледане. Тези курсове са отлична следваща стъпка във вашето развитие като софтуерни инженери и професионалисти от областта на разработката на софтуер.

На къде да продължим след книгата?

Може би се чудите с какво да продължите развитието си като софтуерен инженер? Вие сте поставили с тази книга здрави основи, така че няма да ви в трудно. Можем да ви дадем следните насоки, към които да се ориентирате:

1. Изберете език и платформа за програмиране, например C# + .NET Framework или Java + Java EE или Ruby + Ruby on Rails или PHP + CakePHP. Няма проблем, ако решите да не продължите с езика C#. Фокусирайте се върху технологиите, които платформата ви предоставя, а езикът ще научите бързо. Например ако изберете Objective C и iPhone / iPad / iOS програмиране, придобитото от тази книга алгоритмичното мислене ще ви помогне бързо да навлезете.

2. Прочетете някоя книга за релационни бази данни и се научете да моделирате данните на вашето приложение с таблици и връзки между тях. Научете се как да построявате заявки за извличане и промяна на данните чрез езика SQL. Научете се да работите с някой сървър за бази данни, примерно Oracle, SQL Server или MySQL. Следващата естествена стъпка е да усвоите някоя ORM технология, например ADO.NET Entity Framework, Hibernate или JPA.

3. Научете някоя технология за изграждане на динамични уеб сайтове. Започнете с някоя книга за HTML, CSS, JavaScript и jQuery или с безплатния курс по Web Front-End Development в Академията на Телерик (http://frontendcourse.telerik.com). След това разгледайте какви средства за създаване на уеб приложения предоставя вашата любима платформа, примерно ASP.NET / ASP.NET MVC при .NET платформата и езика C# или Servlets / JSP / JSF при Java платформата или CakePHP / Symfony / Zend Framework при PHP платформата или Ruby on Rails при Ruby или Django при Python. Научете се да правите прости уеб сайтове с динамично съдържание. Опитайте да създадете уеб приложение за мобилни устройства.

4. Захванете се да напишете някакъв по-сериозен проект, например интернет магазин, софтуер за обслужване на склад или търговска фирма. Това ще ви даде възможност да се сблъскате с реалните проблеми от реалната разработка на софтуер. Ще добиете много ценен реален опит и ще се убедите, че писането на сериозен софтуер е много по-трудно от писането на прости програмки.

5. **Започнете работа в софтуерна фирма!** Това е много важно. Ако наистина сте решили всички задачи от тази книга, лесно ще ви предложат работа. Работейки по реални проекти ще научите страхотно много нови технологии от колегите си и ще се убедите, че макар и да знаете много за програмирането, сте едва в началото на развитието си като софтуерен инженер. Само при реална работа по истински проекти в софтуерна фирма съвместно с колеги ще се сблъскате с проблемите при работа в екип и с практиките и инструментите за ефективно преодоляване на тези проблеми. Ще трябва да поработите поне няколко години, докато се утвърдите като специалист по разработка на софтуер. Тогава, може би, ще си спомните за тази книга и ще осъзнаете, че не сте сбъркали започвайки от структурите от данни и алгоритмите вместо директно от уеб технологиите или базите данни.

Безплатни курсове в Академията на Телерик

Можете да си спестите много труд и нерви, ако решите да преминете през всички описани по-горе стъпки от развитието си като софтуерен инженер в Академията на Телерик под ръководството на Светлин Наков и инструктори с реален опит в софтуерната индустрия. Академията е най-лесният и напълно безплатен начин да поставите основите на изграждането си като софтуерен инженер, но не е единственият начин. Всичко зависи от вас!

Ако решите да се възползвате от безплатните курсове по програмиране и софтуерни технологии в Академията на Телерик за софтуерни инженери (присъствено или онлайн), разгледайте курсовете в академията. Към юли 2011 г. това са следните безплатни курсове:

Fundamentals of C# Programming

Курсът следва плътно учебното съдържание на настоящата книга, която е основен учебник към него. Към курса са достъпни за безплатно самообучение лекции, примери, демонстрации, домашни и видеозаписи от лекциите, провеждани в Академията на Телерик.

Успешно завършилите могат да участват в следващите нива от безплатните курсове в Академията на Телерик и да бъдат обучавани за .NET разработчици, QA инженери или специалисти за работа с клиенти.

Курсът се провежда по веднъж всяка година и нови групи започват през есента (септември-октомври).

Официален уеб сайт на курса: http://csharpfundamentals.telerik.com.

.NET Development Essentials

Курсът представлява много задълбочено обучение по разработка на софтуер за платформа .NET Framework с езика C#. Той продължава 5 месеца целодневно и обхваща всички по-важни технологии, които един .NET софтуерен инженер трябва да владее, за да бъде добър професионалист: .NET Framework, бази данни, SQL, SQL Server, ORM технологии, ADO.NET Entity Framework, уеб услуги и WCF, уеб front-end технологии, HTML5, JavaScript, jQuery, ASP.NET, ASP.NET MVC, XAML, WPF, Silverlight, RIA приложения, софтуерно инженерство, design patterns, unit testing, работа в екип и SCRUM.

В края на курса завършващите получават професията ".NET софтуерен инженер" и имат възможност да започнат работа по специалността в Телерик. Учебните материали от курса не са публични.

Курсът се провежда безплатно веднъж годишно и започва през пролетта. В него могат да участват само завършилите с отличие курса "Fundamentals of C# Programming".

Официален уеб сайт на курса: http://dotnetessentials.telerik.com.

Software Quality Assurance and Test Automation (Telerik QA Academy)

Курсът представлява много сериозно и задълбочено обучение по осигуряване на качеството на софтуера и включва както теоретични фундаментални познания за тестването на софтуера, така и практически знания и умения за използване на инструменти за автоматизация на тестването. Курсът обхваща основи на софтуерното тестване, black-box и white-box техники за дизайн на тестове, техники и инструменти за автоматизация на тестовете, тестване на уеб приложения, desktop приложения, уеб услуги и RIA приложения, тестване за натоварване и управление на QA процесите.

Успешно завършилите с добри резултати имат възможност да започнат работа в Телерик като Software Quality Assurance (QA) инженери. Учебните материали от курса не са публични.

Курсът се провежда безплатно веднъж годишно и започва пролетно време. В него могат да участват само завършилите "Fundamentals of C# Programming". Официален уеб сайт: http://qaacademy.telerik.com.

Софтуерна академия за ученици (Telerik School Academy)

Академията на Телерик по софтуерно инженерство за ученици е програма за обучение на ученици от средните училища по разработка на софтуер и софтуерни технологии, която им помага да се подготвят за Националната Олимпиада по Информационни Технологии (НОИТ). Обученията се организират веднъж месечно за 3 дни целодневно. Те са безплатни, но разходите на учениците се поемат от самите тях или от тяхното училище. При наличие на свободни места могат да участват и хора, които не са ученици.

Учебната програма на Академията по софтуерно инженерство за ученици обхваща голямо разнообразие от езици и технологии: езикът за програмиране C#, средата .NET Framework, бази данни и SQL Server, ORM технологии, разработване на front-end приложения с HTML5, JavaScript и jQuery, разработване на уеб приложения с ASP.NET и AJAX, HTML5, разработване на игри, разработване на мобилни приложения, разработване на десктоп приложения с Windows Presentation Foundation (WPF), разработване на RIA приложения със Silverlight. Специално внимание се обръща на подготовката за официалния технически тест на Националната Олимпиада по Информационни Технологии (НОИТ).

Всички учебни материали от проведените обучения са публикувани за свободно изтегляне, а учебните занятия могат да се гледат свободно и като видеозаписи в сайта на академията.

Академията по софтуерно инженерство за ученици се провежда безплатно веднъж на две години (тъй като е доста продължителна). Тя започва есенно време с началото на учебната година в училищата. Официален уеб сайт: http://schoolacademy.telerik.com.

Разработка на уеб Front-End приложения

Курсът дава задълбочени познания и умения за разработка на уеб сайтове и уеб front-end приложения с HTML, CSS, Photoshop, JavaScript, jQuery, работа със CMS системи, HTML 5 и CSS 3. Курсът се препоръчва на всички млади софтуерни инженери, които смятат да се занимават сериозно с уеб технологии. Той се провежда в две части. Първата е насочена към изработката на уеб сайтове (рязане на PSD до XHTML + CSS + картинки), а втората – към разработка на динамични HTML5 front-end приложения с JavaScript, jQuery, AJAX, RESTful Web services и JSON.

Завършилите с отличие получават професията „web front-end developer“ и предложения за работа в ИТ индустрията.

Всички учебни материали от проведените обучения са публикувани за свободно изтегляне, а учебните занятия могат да се гледат свободно и като видеозаписи от сайта на курса.

Курсът се провежда безплатно веднъж годишно и започва пролетно време. Официален уеб сайт: http://frontendcourse.telerik.com.

Разработка на мобилни приложения

Курсът обхваща съвременните технологии за разработка приложения за мобилни устройства. В него се изучават задълбочено технологии за междуплатформена разработка като PhoneGap и разработка за водещи мобилни платформи като Android, iPhone и Windows Phone.

Всички учебни материали от проведените обучения (лекции, упражнения, демонстрации, видеозаписи) се публикуват на сайта на курса.

Курсът се провежда безплатно веднъж годишно и започва есенно време. Официален уеб сайт: http://mobiledevcourse.telerik.com.

Качествен програмен код

Курсът обхваща принципите за изграждане на висококачествен програмен код в процеса на разработка на софтуер. Качеството на кода се разглежда в неговите най-съществени характеристики – коректност, леснота за четене и леснота за поддръжка. Дават се насоки, препоръки и утвърдени практики за конструиране на класове, методи, работа с цикли, работа с данни, форматиране на кода, защитно програмиране и много други. Въвеждат се принципите на компонентно тестване (unit testing) и преработка на кода (refactoring). Наред с теоретичните познания всички участници в курса защитават проект, с който усвояват на практика принципите на качествения код, unit тестването и преработката на лош код.

Всички учебни материали от проведените обучения (лекции, упражнения, демонстрации, видеозаписи) се публикуват на сайта на курса.

Курсът се провежда безплатно веднъж годишно и започва пролетно време. Официален уеб сайт: http://codecourse.telerik.com.

Разработка на уеб приложения с ASP.NET

Курсът въвежда студентите в практическата разработка на съвременни уеб приложения върху платформата Microsoft .NET. Той обхваща основите на езика C#, платформата .NET Framework, базите данни и разработката на уеб приложения с технологиите ASP.NET и AJAX. Студентите научават как да построяват динамични уеб приложения с бази от данни, базирани на ASP.NET, SQL Server и ADO.NET Entity Framework. Основният фокус на учебното съдържание е върху уеб технологиите и уеб програмирането с .NET платформата – започвайки от HTTP, HTML, CSS, JavaScript, през основите на ASP.NET, ASP.NET Web Forms, до по-сложни концепции в ASP.NET (управление на сесия, шаблонни страници, контроли за визуализация на данни, AJAX). Засягат се и теми като мултимедийни приложения (RIA), Silverlight и ASP.NET MVC.

Всички учебни материали от проведените обучения (лекции, упражнения, демонстрации, видеозаписи) се публикуват на сайта на курса.

Курсът се провежда безплатно веднъж годишно и започва есенно време. Официален уеб сайт: http://aspnetcourse.telerik.com.

Успех на всички!

От името на целия авторски колектив ви пожелаваме неспирни успехи в професията и в живота!

Светлин Наков,
Ръководител направление "Технологично обучение", Телерик АД,
Академия на Телерик за софтуерни инженери – http://academy.telerik.com
5.07.2011 г.

Най-голямата ценност на компанията е екипът ѝ от млади и изпълнени с ентусиазъм специалисти

telerik
deliver more than expected

Телерик е водеща компания, предлагаща цялостни решения за разработване и автоматизирано тестване на софтуерни приложения, управление на проекти според agile методологията, бизнес репортинг, както и управление на уеб съдържание, всички от които разработени върху най-новите Microsoft платформи.

Телерик дава възможност на мотивираните, проактивни млади хора, които желаят да се развиват в ИТ индустрията, но нямат необходимите опит и знания, да се обучават безплатно в **АКАДЕМИЯТА НА ТЕЛЕРИК** и да се присъединят към екипа на компанията след успешното ѝ завършване.

Телерик е:

- **Българска иновативна технологична компания, лидер на световния пазар**
- **Златен сертифициран Microsoft партньор**
- **С екип от над 400 служители, повечето от които софтуерни инженери**
- **Работодател #1 за България през 2010 г.**
- **В челната листа на най-добрите работодатели в Централна и Източна Европа**

В Телерик ще откриете:

- **ПОСТОЯННО ОБУЧЕНИЕ И ПОМОЩ, необходими за вашето професионално развитие**
- **ДИНАМИЧНА работна среда и приятелски взаимоотношения**
- **ВЪЗМОЖНОСТ да работите заедно с едни от най-добрите софтуерни специалисти у нас**
- **ОТВОРЕНА и конструктивна комуникация**
- **УСЛОВИЯ и възможност за непрекъснато усъвършенстване**

Best Employer
Central Eastern Europe
2008/2007
conducted by Hewitt, sponsored by The Wall Street Journal Europe

Best Employer
Bulgaria
2010/2009/2008/2007
conducted by Hewitt, sponsored by The Wall Street Journal Europe

Microsoft®
GOLD CERTIFIED
Partner

Microsoft®
Visual Studio®
PARTNER

www.telerik.com/careers

Telerik Academy

Присъедини се към Академията на Телерик!

АКАДЕМИЯТА НА ТЕЛЕРИК предоставя безплатно практическо обучение, насочено към всички млади хора, желаещи да станат умели .NET софтуерни инженери и да се присъединят към екипа на Телерик.

Академията подготвя бъдещите софтуерни специалисти за сложността, разнообразието и динамиката на днешната ИТ действителност, като за целта всички курсисти се обучават да работят с най-новите технологии на Майкрософт и черпят знания и опит от доказали се в областта на програмирането лектори и разработчици на софтуер.

В академията ще получите задълбочени знания и опит, изучавайки:

C#, .NET технологии, бази данни, SQL, Silverlight, WPF, ASP.NET, HTML5, разработка на мобилни приложения за iOS, Android и Windows Phone, основите на софтуерното инженерство

Академията на Телерик ви дава възможност да:

- **Учите напълно БЕЗПЛАТНО**
- **Изберете сред редица РАЗЛИЧНИ КУРСОВЕ**
- **Овладеете ОСНОВИТЕ на софтуерното инженерство**
- **Усвоите ПРОЦЕСА за разработка на софтуер**
- **Получите задълбочени теоретични и практически ИТ ПОЗНАНИЯ**
- **Станете умел .NET СОФТУЕРЕН ИНЖЕНЕР**
- **Започнете своята ИТ кариера в ТЕЛЕРИК – РАБОТОДАТЕЛ #1 в България за 2010 г.**

Само в рамките на две години **АКАДЕМИЯТА НА ТЕЛЕРИК** за софтуерни инженери успя да се наложи като безспорен лидер у нас в предлагането на допълнително обучение за софтуерни специалисти, спомагайки за успешния старт в кариерното развитие на стотици ентусиазирани младежи.

academy.telerik.com

academy@telerik.com *facebook.com/TelerikAcademy*

telerik

deliver more than expected

Telerik Academy SCHOOL

Стани част от Академията на Телерик за ученици!

“Основната цел на обучението е да израстнеш, а нашите умове, за разлика от телата ни, могат да растат през целия ни живот!“ Мортимър Адлър

АКАДЕМИЯТА НА ТЕЛЕРИК за ученици предоставя съвременно, специализирано обучение по софтуерно инженерство за ученици от средните училища в България. Нашата основна цел е да помогнем на младите момчета и момичета да се подготвят за Националната олимпиада по информационни технологии (НОИТ) и да придобият солидни теоретични и практически умения за разработка на софтуер и софтуерни технологии.

В Училищната академия на Телерик по софтуерно инженерство:

- **Стимулираме креативността и мисленето на учениците**
- **Всеки ученик получава индивидуално внимание и съдействие**
- **Полагаме основите за писане на качествен програмен код**
- **Изучаваме съвременните софтуерни технологии**
- **Подготвяме учениците за Националната олимпиада по информационни технологии**
- **Предлагаме голямо разнообразие от практически и теоретични обучения**

Програмата на Академията включва:

C#, .NET технологии, бази данни, SQL, Silverlight, 3D графики и моделиране, разработка на игри, програмиране на микроконтролери, WPF, ASP.NET, HTML5, разработка на мобилни приложения, основи на софтуерното инженерство, работа в екип

schoolacademy.telerik.com

groups.google.com/group/it-olymp *facebook.com/TelerikSchoolAcademy*

telerik

deliver more than expected

БАРС

Българска асоциация
на разработчиците на софтуер

www.devbg.org

Българска асоциация на разработчиците на софтуер (БАРС) е нестопанска организация, която подпомага професионалното развитие на българските софтуерни специалисти чрез образователни и други инициативи.

БАРС работи за насърчаване обмяната на опит между разработчиците и за усъвършенстване на техните знания и умения в областта на проектирането и разработката на софтуер.

Асоциацията организира специализирани конференции, семинари и курсове за обучение по разработка на софтуер и софтуерни технологии.

IT ★ BOXING

http://itboxing.devbg.org

Инициативата "IT Boxing шампионат" събира привърженици на различни софтуерни технологии и технологични доставчици в отворена дискусия на тема "коя е по-добрата технология". По време на тези събирания привърженици на двете технологии, които се противопоставят (примерно .NET и Java), защитават своята визия за по-добрата технология чрез презентации, дискусии и открит спор, който завършва с директен сблъсък с надуваеми боксови ръкавици.

Преди всяко събиране организаторите сформират две групи от експерти, които ще защитават своите технологии. Отборите презентират, демонстрират и защитават своята технология с всякакви средства. Накрая всички присъстващи гласуват и така се определя победителят.

За всички, които се интересуват от безплатни курсове, обучения, семинари и други инициативи, свързани разработката на софтуер и съвременните софтуерни технологии, препоръчваме да следят сайта на д-р Светлин Наков:

www.nakov.com

В него ще намерите:

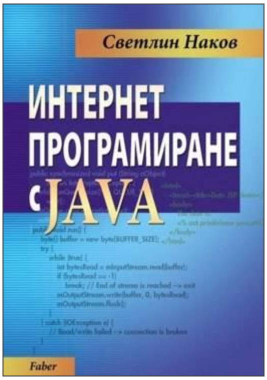
- Покани за безплатни технически курсове, семинари, обучения, видеолекции и презентации
- Технологични новини и статии
- Книгите на Наков и колектив

Светлин Наков
Веселин Колев
и колектив
ВЪВЕДЕНИЕ В ПРОГРАМИРАНЕТО
със C#
telerik
Telerik Academy

Светлин Наков
и колектив
ВЪВЕДЕНИЕ В ПРОГРАМИРАНЕТО
с Java
telerik
SAP Labs Bulgaria
НАРС

Светлин Наков и колектив
ПРОГРАМИРАНЕ ЗА
.net
Framework
1
ТОМ
telerik
НАРС

Светлин Наков и колектив
ПРОГРАМИРАНЕ ЗА
.net
Framework
2
ТОМ
telerik
НАРС


Светлин Наков
ИНТЕРНЕТ ПРОГРАМИРАНЕ
с JAVA
Faber

Светлин Наков
JAVA
Java за цифрово подписване на документи в уеб
НАРС

Съществуват много книги за C# и още повече за програмиране. За много от тях ще кажат, че са най-доброто ръководство, най-бързо навлизане в езика. Тази книга е различна с това, че ще ви покаже какво трябва да знаете, за да постигате успехи. Ако смятате темите в тази книга за безинтересни, вероятно софтуерното инженерство не е за вас.

Веселин Райчев, софтуерен инженер в Google

Досега не съм попадал на книга за програмиране, която едновременно да запознава читателя с езика и да формира уменията му за решаване на задачи. Радвам се, че сега има такава книга, и съм сигурен, че ще бъде изключително полезна на бъдещите програмисти.

Васил Бакалов, софтуерен инженер в Microsoft

Свидетел съм на това какви усилия са положени за написването на тази книга и съм щастлив, че този огромен заряд и желание да се създаде една по-различна книга, наистина се е материализирало в много качествено съдържание.
Книгата ще бъде полезна за читателите като им даде една добра основа, на която да стъпят. Основа, която да ги запали към професионално развитие в областта на програмирането. Това е книга, която ще им помогне да направят един по-безболезнен и качествен старт.

Васил Терзиев, един от основателите и изпълнителен директор на Телерик АД

В книгата ще намерите голяма част от основите на програмирането. Аналогична фундаментална книга в автомобилната индустрия би била озаглавена "Двигатели с вътрешно горене".

Никола Михайлов, софтуерен инженер в Microsoft

С годините можете и сами да стигнете до добрите практики, които тази книга ще ви препоръча, но трябва ли да се учите по метода на пробите и грешките? Тази книга ще ви даде лесния начин да тръгнете в правилната посока – да овладеете базовите структури от данни и алгоритми, да се научите да мислите правилно и да пишете кода си качествено.

Васил Поповски, софтуерен архитект във VMware България

Тази книга не е само за начинаещите. Дори програмисти с няколкогодишен опит има какво да научат от нея. Препоръчвам я на всеки разработчик на софтуер, който би искал да разбере какво не е знаел досега.
Приятно четене!

Любомир Иванов, ръководител отдел "Data and Mobile Applications", Mobiltel

Уеб сайт:
http://introprogramming.info

АВТОРИТЕ:

Веселин Георгиев
Веселин Колев
Дилян Димитров
Илиян Мурданлиев
Йосиф Йосифов
Йордан Павлов
Мира Бивас
Михаил Вълков
Михаил Стойнов
Николай Василев
Николай Костов
Николай Недялков
Павел Дончев
Павлина Хаджиева
Радослав Иванов
Радослав Кирилов
Радослав Тодоров
Светлин Наков
Станислав Златинов
Стефан Стаев
Теодор Божиков
Теодор Стоев
Христо Германов
Цвятко Конов

ISBN 978-954-400-527-6